Taint Inference for Cross-Site Scripting in Context of URL Rewriting and HTML Sanitization

Jinkun Pan, Xiaoguang Mao, and Weishi Li

Currently, web applications are gaining in prevalence. In a web application, an input may not be appropriately validated, making the web application susceptible to cross-site scripting (XSS), which poses serious security problems for Internet users and websites to whom such trusted web pages belong. A taint inference is a type of information flow analysis technique that is useful in detecting XSS on the client side. However, in existing techniques, two current practical issues have yet to be handled properly. One is URL rewriting, which transforms a standard URL into a clearer and more manageable form. Another is HTML sanitization, which filters an input against blacklists or whitelists of HTML tags or attributes. In this paper, we make an analogy between the taint inference problem and the molecule sequence alignment problem in bioinformatics, and transfer two techniques related to the latter over to the former to solve the aforementioned yet-to-be-handled-properly practical issues. In particular, in our method, URL rewriting is addressed using local sequence alignment and HTML sanitization is modeled by introducing a removal gap penalty. Empirical results demonstrate the effectiveness and efficiency of our method.

Keywords: Taint inference, cross-site scripting, URL rewriting, HTML sanitization, bioinformatics.

I. Introduction

Nowadays, accessing web applications has already become a daily routine for many people, such as the checking of emails, conducting bank transactions, and visiting social networking websites. All kinds of information systems for governments, businesses, and individuals are now built as web applications. Unfortunately, many web applications are exposed to various security vulnerabilities. Among them, cross-site scripting (XSS) has emerged as one of the most serious threats on the web. XSS is listed second in the top 10 security risks from OWASP [1], and fourth in the top 25 most dangerous software errors from CWE/SANS [2]. The security problems caused by XSS are of great severity. Through injecting malicious scripts into trusted web contents, an attacker can gain access to a user's browser; steal a user's cookies; hijack a user's sessions; transfer confidential data; cause denial of service; and forge web requests and responses, as well as perform many other types of malicious activities.

Although a single XSS vulnerability is easy to fix, fixing all XSS vulnerabilities in a large web application is a really challenging task, which many application programmers cannot fully accomplish. Instead of fixing them all, detecting and preventing them when they occur is a more feasible way to deal with them. To prevent XSS, we should first detect whether an attacker is able to exert control over a piece of web content, and if so, we should then further detect precisely which parts of this content can the attacker inject into. *Taint inference* is proposed to solve this problem. Such a technique is practically useful on the client side because it works in a manner similar to that of a black box (that is, it compares the input of the user with the response of the server); thus, it does not need source code and is irrelevant to the underlying server technology.

Manuscript received June 24, 2015; revised Oct. 17, 2015; accepted Dec. 9, 2015.

This work was supported by the National Natural Science Foundation of China (Nos. 61379054, 61502015, and 91318301), and Program for New Century Excellent Talents in University.

Jinkun Pan (corresponding author, pan_jin_kun@163.com), Xiaoguang Mao (xgmao@nudt.edu.cn), and Weishi Li (bitasa.student@sina.com) are with the College of Computer, National University of Defense Technology, Changsha, China.

However, some practical issues still remain to be solved. For example, more and more websites are using URL rewriting to overcome the shortcomings of the standard URL — the exposing of the underlying technology of the website, the fact that it is neither descriptive nor friendly to users and search engines alike. URL rewriting impedes existing taint inference techniques in locating and extracting a user input from a URL, thus affecting the precision of any resulting inference. Moreover, many server applications adopt HTML sanitizers to filter potential dangerous tags and attributes in an effort to protect against XSS attacks. This also causes problems when it comes to trying to match the input of a user with the response of the server. To overcome these two problems, we propose a new taint inference technique inspired by molecule sequence alignment in bioinformatics. Through local sequence alignment, a tainted input can be located and inferred automatically within the context of URL rewriting, and the imprecision of taint inference caused by HTML sanitization can be mitigated by introducing a removal gap penalty. We evaluate our technique using 18 vulnerabilities in five open-source projects, each with 108 malicious vectors. Experimental results show that both the inference rate and the inference precision are improved evidently and that the running overhead is negligible.

In the reminder of this paper, we first introduce background techniques in Section II. Then, we describe the motivation for our study in Section III and propose our approach in Section IV. Experimental evaluations are reported in Section V. Finally, we discuss the related work in Section VI and conclude in Section VII.

II. Background

1. Cross-Site Scripting

XSS denotes a kind of code injection attack on a web application. Because HTML documents have a flat, serial structure comprising a mixture of control statements, formatting, and actual content, attackers can inject malicious scripts into the content to be responded to by the vulnerable application, due to a lack of proper input validation and sanitization. As such injected content is delivered from a trusted server, the relevant malicious scripts can act under permissions that are granted to the vulnerable application.

XSS can be classified into three different types: reflected, stored, and DOM-based XSS. A *reflected* XSS vulnerability is the most common type. These vulnerabilities show up when data provided by a web client is used immediately by server-side scripts to parse and display a page of results. A *stored* XSS vulnerability occurs when malicious data provided by an attacker is injected into a vulnerable application's storage. This results in every user that accesses the poisoned web page receiving the injected script without the need for any further action on behalf of the attacker. A *DOM-based* XSS is a special variant of the reflected XSS, where logic errors in legitimate JavaScript and careless usage of client-side data result in XSS conditions. In a DOM-based XSS, malicious data need not touch a web server; rather, it can be reflected by the JavaScript code, fully on the client side.

Regardless of the different types of XSS, the corresponding taint inference algorithms are similar. The differences lie in the contexts and contents to be inferred. To simplify illustration, we only consider reflected XSS in this paper.

2. Taint Inference

XSS attacks occur under the following two conditions:

- Data from an untrusted source is injected into dynamic content that is to be sent to a web user.
- The injected content is able to perform malicious activities — the likes of which is not anticipated by either the developer or the administrator.

To detect the occurrence of an XSS attack, it is necessary to check whether these two conditions have been met. In our research, we focus on the first condition and try to solve the problems of whether data from an untrusted source has been injected and which parts of the response delivered by the server are derived from the injected data.

Fine-grained taint tracking [3]–[7] has been proposed as an effective technique for tackling such problems. However, it suffers from several drawbacks, such as heavy instruments, high overheads, language dependency, and requirement of source codes; thus, these drawbacks make it difficult to adopt such a technique in production systems.

To overcome these drawbacks, a new taint inference technique is proposed, which infers taints using a black-box method by observing and comparing user input requests and server output responses. Generally speaking, requests to web applications use the HTTP protocol, with standardized ways of encoding parameters. Web applications receive the request-related parameter values, apply simple sanitization or normalization operations, and then use the values to retrieve some data, or even generate contents containing these values and respond to the user. As a result, data flows might be identified by comparing input parameter values against all possible substrings of outgoing responses. Because client-side defenses do not (and need not) access the source code, taint inference is preferred rather than taint tracking. An example is shown in Fig. 1. The server code is vulnerable due to a lack of proper input processing. The solid arrows represent a taint flow from the URL request to the HTML response through the

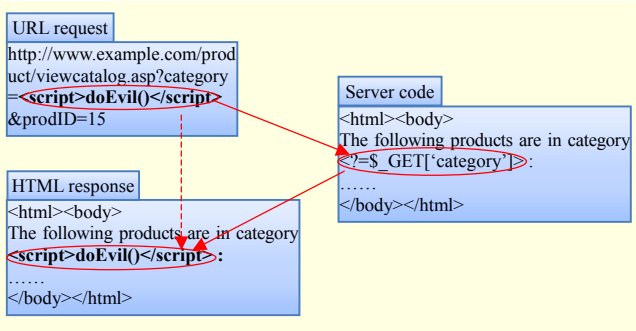

Fig. 1. Example of taint inference in detecting XSS vulnerability.

server code. However, the server code is not available on the client side. Taint inference helps us infer the taint flow between the URL request and the HTML response, shown by the dashed arrow, which discloses the XSS vulnerability concealed in the server code.

In existing techniques, Internet Explorer (IE) [8] uses regular expressions to infer taints. From inputs, regular expressions from possibly malicious injections are created using heuristics. These expressions are then compiled and matched against the HTML output. The taint inference algorithm of XSSAuditor [9] uses the idea of straight string matching between inputs and outputs, considering magic quotes and normalization of unicode characters. NoXSS [10] adopts a longest common subsequence algorithm, which allows parts of a substring to be present in an input parameter while missing in a response. XSSFilt [11] relies on an approximate, rather than exact, string match to be able to identify taint in the presence of simple sanitization or normalization operations used by a web application. These techniques have been proven to be useful in inferring taints that may cause XSS. Nevertheless, there are still some practical issues that need to be further investigated, which will be discussed in the next section.

III. Motivation

1. URL Rewriting

URL rewriting aims to improve the appearance of a given URL. It adds a layer of abstraction between the files used to generate a web page and the URL that is presented to the outside world. Most web servers and web frameworks support URL rewriting, either directly or through extension modules.

Normally, a standard URL looks something like the following:

http://www.example.com/product/viewcatalog.asp?category=shoes&prodID=15

They are prevalent in dynamically generated web pages. However, there are many problems with a URL of this kind:

- It exposes the underlying technology, which gives potential hackers clues as to what they should send along with the query string to perform a front-door attack on the site.
- If the language that the website is based on is changed (to PHP, for instance), all old URLs will stop working.
- The URL is littered with awkward punctuation, such as the question mark and ampersand.
- Many search engines will not index a site in depth if it contains links to such dynamic pages.

Luckily, using rewriting, we can clean up this URL to something far more manageable, such as the following:

http://www.example.com/product/catalog/shoes/15

This URL is more logical, readable, and memorable and will be picked up by search engines. The faux directories are short and descriptive. In addition, it looks more permanent.

Nevertheless, there can be drawbacks as well. A URL is the most prevalent input source of XSS. Existing taint inference techniques rely on standard parameter encoding of URLs to locate and extract user inputs. In such encodings, parameters are located after a question mark and separated by an ampersand (in a URL). Each parameter has a name and a value that are connected with the equals sign. It is easy to parse a standard URL to extract parameter values as the user input to be inferred. However, in the context of URL rewriting, it is hard to extract parameters on the client side, since we do not know the rewriting rules of the server; the only thing we can determine is that parameters may exist in the URL beyond the domain part. Without precise information of the input parameters, the effectiveness of existing taint inference techniques will reduce dramatically. This motivates us to propose a practical method to infer taints without relying on exact URL parameter locations.

2. HTML Sanitation

HTML sanitization is the process of examining an HTML document and producing a new HTML document that preserves only those tags that are deemed to be safe. HTML sanitization can be used to protect against XSS attacks by sanitizing any HTML code submitted by a user. Basic tags for changing fonts are often allowed, such as <b>, <i>, <u>, , and , while more advanced tags such as <script>, <object>, <embed>, and <link> or some attributes of these tags might be removed by the sanitization process. Sanitization is typically performed by using either a whitelist or a blacklist approach. There are a variety of sanitizers for different languages and frameworks, but the principles are the same.

Unfortunately, existing taint inference techniques do not deal with such removal sanitization properly. Specifically, none of them considers the continuous removal region caused by HTML

ETRI Journal, Volume 38, Number 2, April 2016
http://dx.doi.org/10.4218/etrij.16.0115.0570

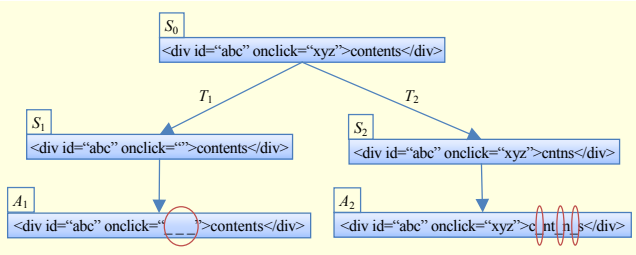

Fig. 2. Example of different transformation cases. Original sequence is S_0. After two different transformations, T_1 and T_2, S_0 becomes S_1 and S_2. Alignments of S_1 and S_2 against S_0 are denoted by A_1 and A_2, respectively, in which "_" is introduced as space caused by removal.

sanitizers. Consider the example shown in Fig. 2. The distance between S_0 and S_1 is the same as that between S_0 and S_2, because the number of "_" introduced are equal. Nevertheless, the transformation from S_0 to S_1 can be done in only one operation, while at least three operations have to happen to transform S_0 into S_2. It is obvious that the former is more plausible and close to the removal operation of HTML sanitizers. However, existing taint inference techniques either reject both S_1 and S_2 due to mismatch, or treat S_1 and S_2 as the same. They do not distinguish continuous removal and separated removal; thus, they are unable to reflect the real distribution of the removed region caused by HTML sanitization. This motivates us to take such a characteristic of HTML sanitization into consideration during taint inference to improve the precision.

IV. Approach

1. Analogy between Taint Inference and Molecule Sequence Alignment

Our approach is mainly inspired by the molecule sequence alignment technique in bioinformatics [12]. Sequence alignment is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. Aligned sequences of nucleotide or amino acid residues are typically represented as rows within a matrix. Spaces are inserted between the residues so that identical or similar characters are aligned in successive columns. If two sequences in an alignment share a common ancestor, then mismatches can be interpreted as point mutations and spaces as *indels* (that is, insertion or deletion mutations) introduced in one or both lineages in the time since they diverged from one another. Sequence alignment has also been used for non-biological sequences, such as those present in natural language or in financial data. By comparing the similarity between sequence alignment and taint inference, we observe an analogy between them, which is shown in Table 1 and detailed in the following sections. By transferring techniques from molecule sequence alignment, we are able to tackle the problem caused by URL rewriting and HTML sanitization.

Table 1. Analogy between sequence alignment and taint inference.

Sequence alignment	Taint inference
Molecule sequence	ASCII character sequence
Semi-global alignment	Standard URL
Local alignment	URL rewriting
Deletion mutation	HTML sanitization
Gap penalty	Removal gap penalty

2. Tackling URL Rewriting by Local Sequence Alignment

There are three kinds of alignment: global, semi-global, and local. In global sequence alignment, an alignment is carried out from beginning until end of a sequence to find out the best possible alignment. However, this is not the case with the taint inference problem, since the HTML response is much longer than the requested URL. Semi-global sequence alignment attempts to find the best possible alignment among the whole of one short sequence and a part of one longer sequence. Existing taint inference techniques are suited to this type of alignment because they assume that a URL is in a standard format and that they can obtain an exact parameter used as a short sequence to be aligned. However, with the presence of URL rewriting, the boundary of such a parameter is no longer clear; thus, semi-global sequence alignment is no longer suited to the situation. In this case, we decide to adopt local sequence alignment to solve the problem. Sequences that are suspected of having similar, or even dissimilar, sequences can be compared by the local alignment method. The method finds local regions that have a high level of similarity. By using local sequence alignment, the precise boundary information of parameters is no longer necessary as it can be inferred during alignment.

There is one key difference that should be noted. In existing taint inference techniques, [11], [13], [14], the *edit distance* is used as a measure, and an object function is then used to minimize the edit distance between two sequences. The edit distance, also referred to as the Levenshtein distance, is the minimum number of edit operations (that is, insertions, deletions, and substitutions) needed to transform one sequence into another. On the contrary, local alignment is defined in

terms of similarity, which maximizes an objective function. When one seeks a pair of substrings to minimize the edit distance, it is often the case that under most natural scoring schemes an optimal pair is a matching pair. However, a substring of a matching pair may only be a single character in length; thus, this is not enough to be able to identify a region of high similarity. A similarity formulation where matches contribute positively and mismatches and spaces contribute negatively is more likely to find more meaningful regions of high similarity. Thus, a similarity scheme rather than a distance scheme is adopted in our taint inference method to handle URL rewriting.

3. Tackling HTML Sanitization by Removal Gap Penalty

Just as a space in an alignment corresponds to an insertion or deletion of a single character, a gap in sequence S_1 opposite substring S_3 in sequence S_2 corresponds to either a deletion of S_3 from S_1 or to an insertion of S_3 into S_2. The concept of a gap in an alignment is therefore important in many biological applications because the insertion or deletion of an entire substring often occurs as a single mutational event, such as unequal crossing-over in meiosis, DNA slippage during replication, and translocations of DNA between chromosomes. This is similar to the case of HTML sanitization.

Since a gap of more than one space can be created by a single removal sanitization, the alignment model should reflect the true distribution of spaces through the use of gaps, not merely the number of spaces in the alignment, as adopted by existing taint inference methods. To accommodate this, we introduce a removal gap penalty — a concept derived from that of the gap penalty found in bioinformatics — but only count the number of deletion gaps considering the removal feature of HTML sanitization. A removal gap penalty is subtracted for each deletion gap that has been introduced. There are different gap penalties; for example, a *gap open* penalty and a *gap extension* penalty. A gap open penalty is always applied at the start of a gap, and then any other gaps following on from this are subject to a gap extension penalty (the lesser of the two penalties). In this way, continuous gaps (representing a removal sanitization) are preferred to separate gaps when searching for similar regions; this helps to detect the part of an input caused by removal. Thus, a removal gap penalty might mitigate the problem caused by HTML sanitization to some extent.

There are different types of gap weight models; for example, constant, affine, convex, and arbitrary gap models. Considering factors such as effectiveness, efficiency, and difficulty of computation, the affine gap weight model is the most commonly used gap model in the biology domain. Therefore, we adopted the affine gap weight model in our algorithm as well.

4. Algorithm

The Smith–Waterman algorithm [15] is a well-known dynamic programming algorithm for performing local sequence alignment for determining similar regions between two DNA or protein sequences. In [16], an affine gap model is introduced to the Smith–Waterman algorithm. We adopt such an algorithm and modify it so as to be fit for the case of taint inference. In particular, we only consider a deletion gap instead of both an insertion gap and a deletion gap. Moreover, we use a substitution matrix that consists of ASCII characters to replace the one with nucleotide or amino acids. Here, we present the main idea of our taint inference algorithm using local sequence alignment with removal gap penalty (LSARGP).

To align an input URL, S_{in}, and an output response, S_{out}, consider the prefixes $S_{in}[1, \dots, i]$ of S_{in} and $S_{out}[1, \dots, j]$ of S_{out}. The following three categories can be used to characterize all possible alignments of such prefixes:

1) Character $S_{in}(i)$ is aligned to a character strictly of the left of character $S_{out}(j)$. Therefore, the alignment ends with a gap in S_{in}, which is the case of insertion.
2) Character $S_{in}(i)$ is aligned strictly to the right of $S_{out}(j)$. Therefore, the alignment ends with a gap in S_{out}, which is the case of deletion.
3) Character $S_{in}(i)$ and $S_{out}(j)$ are aligned opposite each other. This includes both the case that $S_{in}(i) = S_{out}(j)$ and that $S_{in}(i) \neq S_{out}(j)$, representing match and mismatch, respectively.

We define $E(i, j)$, $F(i, j)$, and $G(i, j)$ as the maximum score value of any alignment of the above three types, respectively, and $V(i, j)$ as the optimal alignment score of two prefixes. Assume the length of S_{in} is m, and the length of S_{out} is n. The dynamic programming solves the original problem by dividing the problem into smaller independent sub-problems with the following three steps:

1) Initialization of the matrix.

$$\begin{cases} V(i,0)=E(i,0)=0 & 0\le i\le m, \\ V(0,j)=F(0,j)=0 & 0\le j\le n. \end{cases}$$

The initialization of zero for each row and column allows the local alignment to start from any position.

2) Matrix filling with the appropriate scores.

To fill each cell, we should know the neighbor values (diagonal, upper, and left) of the current cell. Define W_m, W_{ms}, and W_g as the score weight of match, mismatch, and gap, respectively. The objective is to find an alignment to

$$\text{maximize}\left[W_m(\#matches)-W_{ms}(\#mismatches)-W_g(\#gaps)\right].$$

In the affine gap weight model, the weight contributed by a single gap of length q is given by the affine function $W_g = W_b + qW_e$, where W_b is the weight of the gap open penalty

ETRI Journal, Volume 38, Number 2, April 2016
http://dx.doi.org/10.4218/etrij.16.0115.0570

and W_e is the weight of the gap extension penalty. The equation $W_g(q+1) - W_g(q) = W_e$ holds for any gap length q greater than zero. Therefore, when evaluating $F(i, j)$, we need not be concerned with where a gap begins, but only whether it has already begun or whether a new gap is being started. The general recurrences are

$$E(i,j) = \max \begin{cases} E(i,j-1) - W_b \\ V(i,j-1) - W_b \end{cases} \quad \text{for } 1 \le i \le m,\ 1 \le j \le n,$$

$$F(i,j) = \max \begin{cases} F(i-1,j) - W_e \\ V(i-1,j) - W_b \end{cases} \quad \text{for } 1 \le i \le m,\ 1 \le j \le n,$$

$$G(i,j) = \max \begin{cases} V(i-1,j-1) + W_m & \text{if } S_{in}(i) = S_{out}(j) \\ V(i-1,j-1) - W_{ms} & \text{if } S_{in}(i) \ne S_{out}(j) \end{cases}, \quad 1 \le i \le m,\ 1 \le j \le n,$$

$$V(i,j) = \max \begin{cases} 0 \\ E(i,j) \\ F(i,j) \\ G(i,j) \end{cases} \quad \text{for } 1 \le i \le m,\ 1 \le j \le n.$$

The zero considered in $V(i, j)$ allows as to say that the best alignment between two prefixes is the empty alignment and just start over. After filling the matrix, a pointer points to a cell that is filled previously, from where the maximum score has been determined.

3) Trace back the two sequences that are being compared and aligned for a suitable alignment.

The final step for the appropriate alignment is *trace backing*. Prior to that, we should find out the maximum score obtained in the entire matrix. The trace back begins from the position that has the highest value, pointing back with the pointers; thus, find out the possible predecessor, and then move to the next predecessor, and continue in this fashion until we reach the score zero.

Examination of the recurrences shows that for any pair (i, j), each of the terms $E(i, j)$, $F(i, j)$, $G(i, j)$, and $V(i, j)$ is evaluated by a constant number of references to previously computed values, arithmetic operations, and comparisons. Hence, $O(mn)$ time suffices to fill in the $(m + 1)(n + 1)$ cells in the dynamic programming table. Therefore, the optimal local alignment with affine gap weights can be computed in $O(mn)$ time.

V. Experiment

1. Research Questions

Through our experiments, we would like to answer the following research questions:

1) Is our method effective in the context of URL rewriting?
2) Is our method effective in the context of HTML sanitization?
3) Is our method efficient in practical deployment?

2. Experiment Setup

We used Apache as the web server. It has its own built-in URL rewriting module called "mod_rewrite." HTML Purifier, a standards-compliant HTML filter library written in PHP, was adopted as the HTML sanitizer. HTML Purifier comes with a thoroughly audited, secure (yet permissive) whitelist. Besides, it supports user-defined lists, thus facilitating us to evaluate different levels of sanitization.

The subject projects were collected from five open-source PHP-based web applications of different sizes ranging from 2k LOC to 44k LOC with 18 vulnerabilities in total. Table 2 shows the detailed information. The vulnerability information is known to the public and accessible from various security advisories, such as Bug-Traq, CVE, and PMASA. The projects were obtained from SourceForge. All of them have been used in evaluating some vulnerability detection approaches previously. Each vulnerable page was tested on 108 tricky and obfuscated HTML-based attack vectors from "XSS cheat sheet" [17] — a well-known and often-cited source for XSS filter circumvention techniques. Thus, there are $18 \times 108 = 1{,}944$ pairs of input and output to be inferred for each setting.

As a comparison, we also evaluated the performance of a taint inference algorithm adopted by XSSFilt using an approximate substring match by edit distance (ASMED). It was chosen because its approximate nature accommodates the existence of URL rewriting and HTML sanitization to some extent. Moreover, its running efficiency is good enough to put

Table 2. Information of subject projects.

Subject project	Description	LOC	Security advisories	# of vulnerabilities
FaqForge 1.3.2	Tool for document management	2238	Bugtraq-43897	4
webChess 0.9.0	Online chess game	3236	Bugtraq-43895	8
SchoolMate 1.5.4	Tool for school administration	8145	groups.csail.mit.edu/pag/ardilla/	3
Phorum 5.2.18	Message board application	12324	CVE-2011-4561	1
PhpMyAdmin 3.4.4	Database management for MySQL	44628	PMASA-2011-16	2

Table 3. Evaluation metrics.

Metric	Description	Calculation
Inference rate	Percentage of instances that are inferred correctly among all instances	$\frac{c}{n}$, where c refers to the number of correctly inferred sequences and n refers to the total number of instances in the dataset
Inference precision	Proportion of the correctly inferred parts, which is more fine-grained compared to the binary criterion that whether a sequence is inferred correctly or not	$\frac{2\times\text{len(Overlap)}}{\text{len(Correct)}+\text{len(Inferred)}}$, where Correct refers to the correct sequence to be inferred and "Inferred" refers to the sequence inferred by the taint inference algorithm, "Overlap" refers to the largest common part between "Correct" and "Inferred," and "len" is the function computing the length of a sequence
Mean	Average value of the inference precision	Sum of all inference precision values divided by the size of the dataset
Median	Middle value of the inference precision	Value in the center of all inference precision values in ascending order
p-value	Probability of falsely rejecting the null hypothesis	$P\{\|U\|>V\}$, where U is the Mann-Whitney U test statistics and V is the sample statistics of U in the dataset
Effect size	Magnitude of the difference between two compared methods	$\frac{R}{n^2}-\frac{n+1}{2n}$, where R is the rank sum in the Mann-Whitney U test

into practical use, as opposed to some complex algorithms such as longest common sequence. In addition, it does not need predefined rules such as regular expressions crafted in advance. In ASMED, the distance of each edit operation is set to be the same. In our algorithm, LSARGP, the match score, the mismatch penalty, and the gap open penalty are set to be the same: furthermore, the gap extend penalty is set to be one-tenth of the gap open penalty. We evaluated several different values of gap extend penalty and found the one-tenth of the gap open penalty to be simple and good enough. Therefore, we only report results with it due to space limitations.

To assess the performance of taint inference techniques, we adopted the metrics shown in Table 3. A Mann–Whitney U test [18] and an A-test [19], two nonparametric statistical approaches, were used to measure the significance of any differences between ASMED and LSARGP. In the case of the Mann–Whitney U test, the null hypothesis (H_0) is that data from the two algorithms share the same distribution; the alternate hypothesis (H_1) is that they have different distributions. Any difference between the two distributions is statistically significant when the null hypothesis is rejected at a significance level of 5%. To further assess the difference quantitatively, we use the nonparametric Vargha–Delaney A-test, which is recommended in [20], to evaluate the magnitude of any difference by measuring the *effect size* of inference precision. In the case of the A-test, the bigger the deviation of the effect size from a value of 0.5, the greater the magnitude of any difference between the two groups studied.

We ran all the experiments on a desktop computer with an Intel Core i7 3.5 GHz processor and 4 GB RAM using Ubuntu 14.04 LTS. The algorithms were implemented in Python 2.7.

Table 4. Examples of different URL formats.

URL	Example
URL0	http://www.example.com/product/viewcatalog.asp?category=**shoes**&prodID=**15**
URL1	http://www.example.com/product/viewcatalog.asp?**category=shoes_prodID=15**
URL2	http://www.example.com/**product/viewcatalog/category/shoes/prodID/15**
URL3	http://www.example.com/**product/viewcatalog/shoes/15**

3. Experimental Results

A. URL Rewriting

First, we evaluated the performance of our taint inference in the context of URL rewriting. Details of the different URL formats evaluated are shown in Table 4. The part to be inferred is in bold. The rewritten part is enlarged from URL0 to URL3. URL0 is the standard URL without rewriting, and the parameters can be extracted precisely. URL1 changes the parameter separator from "&" to "_"; thus, the whole of the part after "?" should be inferred as a whole. URL2 removes the suffix of the dynamic page and replaces the "?" and "&" with "/." URL3 further omits the parameter names based on URL2. For URL2 and URL3, all the parts after the domain should be inferred.

The results of inference rate are shown in Fig. 3(a). Both algorithms perform well without URL rewriting. However, ASMED cannot infer the exact content so long as a semblance of URL rewriting exists, whereas LSARGP can handle around

ETRI Journal, Volume 38, Number 2, April 2016
http://dx.doi.org/10.4218/etrij.16.0115.0570

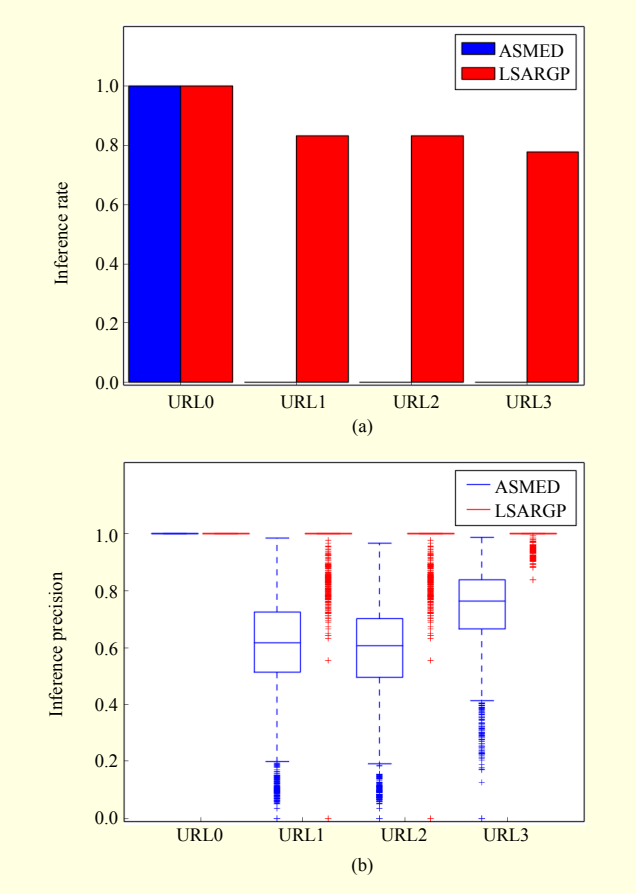

Fig. 3. Results of different URLs: (a) histogram for inference rate and (b) boxplot for inference precision.

Table 5. Taint precision statistics of different URLs.

URL	Inference algorithm	Mean	Median	*p*-value	Effect size
URL0	ASMED	1.00	1.00	0.499994	0.50
	LSARGP	1.00	1.00		
URL1	ASMED	0.59	0.62	0.000000	0.98
	LSARGP	0.97	1.00		
URL2	ASMED	0.57	0.61	0.000000	0.98
	LSARGP	0.97	1.00		
URL3	ASMED	0.72	0.76	0.000000	0.99
	LSARGP	0.99	1.00		

80% of the cases for each format. Figure 3(b) shows the results of inference precision, and Table 5 gives the statistical comparisons. As we can see from the boxplot, the precision of LSARGP is much better than ASMED in URL1, URL2, and URL3. The *p*-values of them are zero, demonstrating the results are of statistical significance. The effect sizes are close to one, which shows the distinct advantage of LSARGP over ASMED.

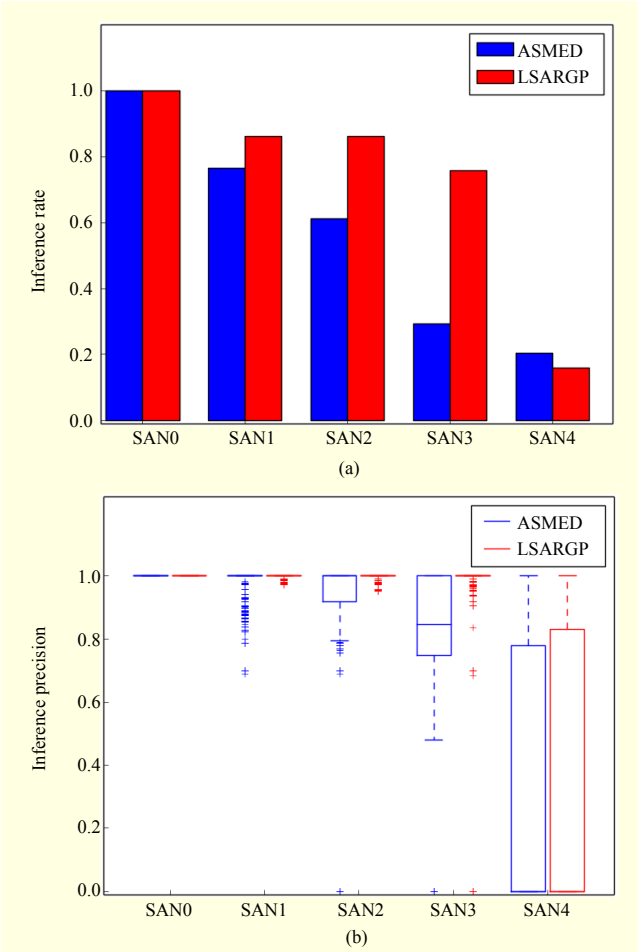

Fig. 4. Results of different sanitizers: (a) histogram for inference rate and (b) boxplot for inference precision.

From the results, we can answer research question 1) in as far as saying that our method handles the URL rewriting effectively.

B. HTML Sanitization

To evaluate the impact of HTML sanitization, we used sanitization rules of different strict levels. From SAN0 to SAN4, the level of strictness increases, which means more parts of the content might be removed. SAN0 is the case without any sanitization. SAN1 filters "script" tags and attributes that can trigger scripts, such as "onclick." SAN2 further removes attributes like "src" or "href," which might include remote scripts. SAN3 uses a more sophisticated whitelist from HTML Purifier, which sanitizes more potentially dangerous tags and attributes. SAN4 removes all HTML tags, which is rarely in reality for the sake of usability and just included as an extreme case.

The results of different sanitizations are shown in Fig. 4 and Table 6. Both algorithms perform well without any sanitization. Their respective performances decrease along with the increase in the level of strictness. In extreme cases (for example, where

Table 6. Taint precision statistics of different sanitizers.

Sanitizer	Inference algorithm	Mean	Median	*p*-value	Effect size
SAN0	ASMED	1.00	1.00	0.499994	0.50
	LSARGP	1.00	1.00		
SAN1	ASMED	0.97	1.00	0.000000	0.56
	LSARGP	1.00	1.00		
SAN2	ASMED	0.95	1.00	0.000000	0.65
	LSARGP	1.00	1.00		
SAN3	ASMED	0.78	0.84	0.000000	0.78
	LSARGP	0.95	1.00		
SAN4	ASMED	0.32	0.00	0.193539	0.49
	LSARGP	0.31	0.00		

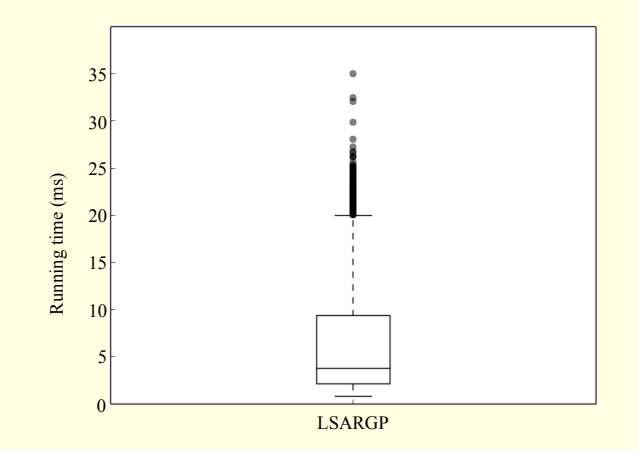

Fig. 5. Boxplot of running time.

all HTML tags are disallowed), both algorithms perform poorly, because of large parts of removal; thus, there is a lack of adequate information for inference.

Nevertheless, from Fig. 4(a), we can see that in common cases such as those of SAN1, SAN2, and SAN3, LSARGP can infer correctly in more than 75% of cases, which is better than that of ASMED; this advantage increases along with an increase in the level of strictness. The advantage of taint precision is also conspicuous, which is shown in Fig. 4(b) and Table 5. The *p*-values are zero and the effect sizes are above 0.5 in SAN1, SAN2, and SAN3. Moreover, the effect size increases as with the level of strictness. The advantage of LSARGP is most conspicuous in SAN3, which is the case of the most practical setting. From the results, we can conclude that LSARGP performs well in practical cases, which answers research question 2) in as far as to say that our method is effective in dealing with HTML sanitization.

C. Running Time

To evaluate the running overhead of our taint inference method, we recorded the running time of the algorithm in all runs of previous experiments and report them in Fig. 5. The mean time is just around 6 ms and the maximum time cost is only 35 ms, which is negligible compared to the response time of a web page. Thus, we can answer research question 3) in as far as we can say that our method is efficient enough to be put into practical use.

VI. Related Work

XSS vulnerabilities have received a great deal of attention in research, and a variety of approaches have been proposed to tackle them. Different from general program bugs, XSS cannot be discovered by common debugging techniques [21], [22]. Detecting XSS and defending against it at runtime is a more practical way. Earlier works have been mostly in the form of server-side defenses. XSSDS [10] is based on passive HTTP traffic monitoring. Reflected XSS attacks can be detected through string matching, and stored XSS attacks can be detected by establishing a set whitelist of scripts. Reference [7] uses string matching and taint-aware policies to stop generic injection attacks. It uses precise taint tracking, and the policies are based on syntactic confinement. XSS-GUARD [23] works by dynamically learning the set of scripts that a web application intends to create for any HTML request. Blueprint [24] converts the untrusted HTML embedded in a page into JavaScript code to fix the browser's interpretation of the page at the server side. Reference [25] proposes an efficient black-box taint inference technique. Taint is inferred by an approximate substring match. It also proposes a flexible syntax- and taint-aware policy framework. Client-side approaches protect users against XSS vulnerabilities without waiting for websites to fix them. IE [8] comes with built-in XSS protection. It first marks requests that look suspicious. Responses to such requests are then scanned for script content that may be derived from suspicious parameters, and this content is then sanitized to prevent its interpretation as a script. XSSAuditor [9] proposes that a new architecture mediate between the HTML parser and the JavaScript engine, which achieves both high performance and high precision. NoScript [26] includes an XSS filter, in which regular expressions are used to extract and identify malicious data from a URL. Reference [27] presents a modification to Firefox's JavaScript engine that prevents data leaks using fine-grained dynamic taint tracking on the client side, refusing to transfer sensitive information to third parties. XSSFilt [11] can detect partial script injections, and uses approximate rather than exact string matching to detect reflected content. Hybrid approaches are an alternative, in which the server is responsible for identifying untrusted data

that it reports to the browser, and a modified browser ensures that XSS attacks cannot result from parsing the untrusted data. BEEP [28] allows the server to supply a policy for the page through a JavaScript function. Nonespaces [29] is an end-to-end mechanism that allows a server to identify untrusted content and reliably convey this information to the client, as well as allowing the client to enforce a security policy on the untrusted content. DSI [30] enforces the structure integrity of web documents through parser-level isolation of untrusted data in the browser based on a server-specified policy. Content Security Policy [31] is a technique to support server-supplied content restrictions by specifying a list of trusted hosts allowed to provide content for the web page.

VII. Conclusion

In this paper, we proposed and evaluated a new taint inference technique to infer taints for XSS within the context of URL rewriting and HTML sanitization. By making an analogy between the taint inference problem and the molecule sequence alignment problem in bioinformatics, these two domains are shown to be connected; the results of our empirical evaluation confirm such a connection. As a result, we believe that researchers in the field of taint inference should look to be inspired by and learn more from the ever-developing techniques in bioinformatics. For example, a more sophisticated substitution matrix can be introduced to reflect the distribution of different characters and the probability of transformation performed by the server application, which will be our future work.

References

[1] OWASP, *The Ten Most Critical web Application Security Risks*, OWASP, 2013. Accessed Jan. 15, 2016. http://www.owasp.org/index.php/Top_10_2013-Top_10

[2] B. Martin et al., *2011 CWE/SANS Top 25 Most Dangerous Software Errors*, The MITRE Corporation, 2011. Accessed Jan. 15, 2016. http://cwe.mitre.org/top25/

[3] S. Chen et al., "Defeating Memory Corruption Attacks via Pointer Taintedness Detection," *Int. Conf. Dependable Syst. Netw.*, Yokohama, Japan, June 28–July 1, 2005, pp. 378–387.

[4] G.E. Suh et al., "Secure Program Execution via Dynamic Information Flow Tracking," *ACM SIGPLAN Notices*, vol. 39, no. 11, Nov. 2004, pp. 85–96.

[5] W.G. Halfond et al., "Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks," *ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Portland, OR, USA, Nov. 5–11, 2006, pp. 175–185.

[6] L.C. Lam and T.-C. Chiueh, "A General Dynamic Information Flow Tracking Framework for Security Applications," *Annual Comput. Security Appl. Conf.*, Miami, FL, USA, Dec. 11–15, 2006, pp. 463–472.

[7] W. Xu, S. Bartkar, and R. Sekar, "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks," *Conf. Usenix Security*, Vancouver, Canada, July 31–Aug. 3, 2006, pp. 121–136.

[8] D. Ross, *IE 8 XSS Filter Architecture/Implementation*, Microsoft Security Research and Defense Blog, 2008. Accessed Jan. 15, 2016. http://blogs.technet.com/srd/archive/2008/08/18/ie-8-xss-filter-architecture-implementation.aspx

[9] D. Bates, A. Barth, and C. Jackson, "Regular Expressions Considered Harmful in Client-Side XSS Filters," *Int. Conf. World Wide Web*, Raleigh, NC, USA, Apr. 26–30, 2010, pp. 91–100.

[10] M. Johns, B. Engelmann, and J. Posegga, "Xssds: Server-Side Detection of Cross-Site Scripting Attacks," *Annu. Comput. Security Appl. Conf.*, Anaheim, CA, USA, Dec. 8–12, 2008, pp. 335–344.

[11] R. Pelizzi and R. Sekar, "Protection, Usability, and Improvements in Reflected XSS Filters," *ACM Symp. Inf. Comput. Commun. Security*, Seoul, Rep. of Korea, May 2–4, 2012, pp. 5–15.

[12] D. Gussfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computional Biology*, Cambrigde, UK: The Press Syndicate of the University of Cambridge, 1997, pp. 215–245.

[13] F. Duchene et al., "LigRE: Reverse-Engineering of Control and Data Flow Models for Black-Box XSS Detection," *Work. Conf. Reverse Eng.*, Koblenz, Germany, Oct. 14–17, 2013, pp. 252–261.

[14] F. Duchene et al., "KameleonFuzz: Evolutionary Fuzzing for Black-Box XSS Detection," *ACM Conf. Data Appl. Security Privacy*, San Antonio, TX, USA, Mar. 3–5, 2014, pp. 37–48.

[15] T.F. Smith and M.S. Waterman, "Identification of Common Molecular Subsequences," *J. Molecular Biology*, vol. 147, no. 1, Mar. 1981, pp. 195–197.

[16] O. Gotoh, "An Improved Algorithm for Matching Biological Sequences," *J. Molecular Biology*, vol. 162, no. 3, Dec. 1982, pp. 705–708.

[17] R. Hansen. *XSS Filter Evasion Cheat Sheet*, OWASP, 2016. Accessed Jan. 15, 2016. http://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

[18] F. Wilcoxon, "Individual Comparisons by Ranking Methods," *Biometrics Bulletin*, vol. 1, no. 6, Dec. 1945, pp. 80–83.

[19] A. Vargha and H.D. Delaney, "A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong," *J. Educational Behavioral Stat.*, vol. 25, no. 2, June 2000, pp. 101–132.

[20] A. Arcuri and L. Briand, "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering," *Int. Conf. Softw. Eng.*, Waikiki, HI, USA, May 21–

18, 2011, pp. 1–10.
[21] Y. Lei et al., "Effective Fault Localization Approach Using Feedback," *IEICE Trans. Inf. Syst.*, vol. 95D, no. 9, Sept. 2012, pp. 2247–2257.
[22] X. Mao et al., "Slice-Based Statistical Fault Localization," *J. Syst. Softw.*, vol. 89, Mar. 2014, pp. 51–62.
[23] P. Bisht and V. Venkatakrishnan, "XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks," *Detection Intrusions Malware, Vulnerability Assessment*, Paris, France, July 10–11, 2008, pp. 23–43.
[24] M.T. Louw and V. Venkatakrishnan, "Blueprint: Robust Prevention of Cross-Site Scripting Attacks for Existing Browsers," *IEEE Symp. Security Privacy*, Oakland, CA, USA, May 17–20, 2009, pp. 331–346.
[25] R. Sekar, "An Efficient Black-Box Technique for Defeating Web Application Attacks," *Annual Netw. Distrib. Syst. Security Symp.*, San Diego, CA, USA, Feb. 8–11, 2009, pp. 21–37.
[26] G. Maone, *NoScript-JavaScript/Java/Flash blocker for a safer Firefox experience*, InformAction, 2012. Accessed Jan. 15, 2016. https://noscript.net/
[27] P. Vogt et al., "Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis," *Annual Netw. Distrib. Syst. Security Symp.*, San Diego, CA, USA, Feb. 28–Mar. 2, 2007, pp. 37–48.
[28] T. Jim, N. Swamy, and M. Hicks, "Defeating Script Injection Attacks with Browser-Enforced Embedded Policies," *Int. Conf. World Wide Web*, Banff, Canada, May 8–12, 2007, pp. 601–610.
[29] M. Van Gundy and H. Chen, "Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks," *Annual Netw. Distrib. Syst. Security Symp.*, San Diego, CA, USA, Feb. 8–11, 2009, pp. 38–55.
[30] Y. Nadji et al., "Document Structure Integrity: A Robust Basis for Cross-Site Scripting Defense," *Annual Netw. Distrib. Syst. Security Symp.*, San Diego, CA, USA, Feb. 8–11, 2009, pp. 1–20.
[31] S. Stamm, B. Sterne, and G. Markham, "Reining in the Web with Content Security Policy," *Int. Conf. World Wide Web*, Raleigh, NC, USA, Apr. 26–30, 2010, pp. 921–930.

Jinkun Pan received his BS and MS degrees in computer science and technology from the National University of Defense Technology (NUDT), Changsha, China, in 2010 and 2012, respectively. He is now a PhD candidate in software engineering at the College of Computer, NUDT. His main research interests include web security, client-side vulnerability, and malicious script detection.

Xiaoguang Mao received his PhD degree in computer science from the National University of Defense Technology (NUDT), Changsha, China, in 1997. He is now a professor at the College of Computer and Laboratory of Science and Technology on Integrated Logistics Support, NUDT. His research interests include high confidence software, software maintenance, and automated program repair.

Weishi Li received his MS degree in computer science and technology from the National University of Defense Technology (NUDT), Changsha, China, in 2011. He is now a PhD candidate in software engineering at the College of Computer, NUDT. His main research interests include fault localization and automated program repair.