# Malicious URL Detection by Dynamically Mining Patterns without Pre-defined Elements *

Da Huang[†‡], Kai Xu[‡], and Jian Pei[†]

[†] Simon Fraser University, {jpei, huangdah}@sfu.ca
[‡] Fortinet Inc. kxu@fortinet.com

**Abstract.** Detecting malicious URLs is an essential task in network security intelligence. In this paper, we make two new contributions beyond the state-of-the-art methods on malicious URL detection. First, instead of using any pre-defined features or fixed delimiters for feature selection, we propose to dynamically extract lexical patterns from URLs. Our novel model of URL patterns provides new flexibility and capability on capturing malicious URLs algorithmically generated by malicious programs. Second, we develop a new method to mine our novel URL patterns, which are not assembled using any pre-defined items and thus cannot be mined using any existing frequent pattern mining methods. Our extensive empirical study using the real data sets from Fortinet, a leader in the network security industry, clearly shows the effectiveness and efficiency of our approach.

## 1 Introduction

A web threat refers to any threat that uses the internet to facilitate cybercrime [15]. In practice, web threats may use multiple types of malware and fraud. A common feature is that web threats all use HTTP or HTTPS protocols, though some threats may additionally use other protocols and components, such as links in emails or IMs, or malware attachments. Through web threats, cyber-criminals often steal private information or hijack computers as bots in botnets. It has been well realized that web threats lead to huge risks, including financial damages, identity thefts, losses of confidential information and data, thefts of network resources, damaged brand and personal reputation, and erosion of consumer confidence in e-commerce and online banking. For example, Gartner [4] estimated that phishing attacks alone caused 3.6 million adults losing 3.2 billion US dollars in the period from September 2006 to August 2007, and the global cost of spam in 2007 was about 100 billion US dollars. The cost of

* We are deeply grateful to the anonymous reviewers for their insightful and constructive comments and suggestions that help to improve the quality of this paper. The work is supported in part by an NSERC Discovery Grant, and NSERC ENGAGE Grant, and a BCFRST NRAS Endowment Research Team Program project. All opinions, findings, conclusions and recommendations in this paper are those of the author and do not necessarily reflect the views of the funding agencies.

spam management in the U.S. alone was estimated 71 billion dollars in 2007 [5]. In Web 2.0 applications, users are even more vulnerable to web threats due to the increasing online interactivity.

Although the exact adversary mechanisms behind web criminal activities may vary, they all try to lure users to visit malicious websites by clicking a corresponding URL (Uniform Resource Locator). A URL is called *malicious* (also known as *black*) if it is created in a malicious purpose and leads a user to a specific threat that may become an attack, such as spyware, malware, and phishing. Malicious URLs are a major risk on the web. Therefore, detecting malicious URLs is an essential task in network security intelligence.

In practice, malicious URL detection faces several challenges.

- *Realtime detection.* To protect users effectively, a user should be warned before she/he visits a malicious URL. The malicious URL detection time should be very short so that users would not have to wait for long and suffer from poor user experience.
- *Detection of new URLs.* To avoid being detected, attackers often create new malicious URLs frequently. Therefore, an effective malicious URL detection method has to be able to detect new, unseen malicious URLs. In practice, the capability of detecting new, unseen malicious URLs is of particular importance, since emerging malicious URLs often have high hit counts, and may cause serious harms to users.
- *Effective detection.* The detection should have a high accuracy. When the accuracy is of concern, the visit frequency of URLs should also be considered. From a user's point of view, the accuracy of a detection method is the number of times that the detection method classifies a URL correctly versus the number of times that the method is consulted. Please note that a URL may be sent to a detection method multiple times, and should be counted multiple times in the accuracy calculation. Therefore, detecting frequently visited URLs correctly is important.
  Similarly, it is highly desirable that a malicious URL detection method should have a high recall so that many malicious URLs can be detected. Again, when recall is calculated in this context, the visit frequency of URLs should be considered.

To meet the above challenges, the latest malicious URL detection methods try to build a classifier based on URLs. *A foundamental assumption is that a clean training sample of malicious URL and good URL samples is available.* Such methods segment a URL into tokens using some delimiters, such as "/" and "?", and use such tokens as features. Some methods also extract additional features, such as WHOIS data and geographic properties of URLs. Then, machine learning methods are applied to train a classification model from the URL sample.

Although the existing works report good performance on classification of malicious URLs by using machine learning algorithms, the result of these machine learning algorithms is not human interpretable. However, in real applications, the human interpretable malicious URL patterns are highly desirable because:

1. In malicious URL detection, people may not only want to catch new malicious URLs, but also be interested in identifying the campaigns behind the malicious activities. For example, in anti-phishing system, detecting a URL is phishing is usually not enough, how to identify the phishing targets is also very important. The URL patterns can exactly play such a role to help people analyze the malicious activities. For example, the URL patterns "`*paypal*.cmd.*/*/login.php`", "`*paypal*.webscr.*`", and "`*paypal*.cmd.*/webscr*`" clearly indicate the phishing campaigns aims at paypal. The patterns can be further used to analyze the malicious activities trends.
2. In real applications, the training data used to train the detection model may be biased and contain some noises. In such cases, the human intervention is very important. The human interpretable URL patterns can be easily modified and adapted by network security experts, which can highly prompt the detection quality.

Thus, in this paper, we introduce such a URL pattern mining algorithm, and we make two new contributions beyond the state-of-the-art methods on malicious URL detections. First, instead of using any pre-defined features or fixed delimiters for feature selection, we propose to dynamically extract lexical patterns from URLs. Our novel model of URL patterns provides new flexibility and capability on capturing malicious URLs algorithmically generated by malicious programs. The patterns extracted by our algorithm are human interpretable, and these patterns can help people analyze the malicious activities. For example, our method can extract patterns like "`*paypal.*.cgi.*/login.php`" where $*$ is a wildcard symbol. Second, we develop a new method to mine our novel URL patterns, which are not assembled using any pre-defined items and thus cannot be mined using any existing frequent pattern mining methods. Our extensive empirical study using the real data sets from Fortinet, a leader in the network security industry, clearly shows the effectiveness and efficiency of our approach.

The rest of the paper is organized as follows. In Section 2, we review the state-of-the-art methods and point out how our method is different from them. We discuss our lexical feature extraction method in Section 3, and devise our pattern mining method in Section 4. We report our empirical study results in Section 5, and conclude the paper in Section 6.

## 2 Related Work

The existing work on malicious URL detection can be divided into three categories, namely the blacklist based methods, the content based methods, and the URL based methods.

The *blacklist based methods* maintain a blacklist of malicious URLs. During detection, a URL is reported as malicious if it is in the blacklist. Most of the current commercial malicious URL detection systems, such as Google Safebrowsing (`http://www.google.com/tools/firefox/`

safebrowsing/index.html), McAfee SiteAdvisor (http://www.siteadvisor.com), Websense ThreatSeeker Network (http://www.websense.com/content/threatseeker.asp), and Fortinet URL lookup tool (http://www.fortiguard.com/ip_rep.php), use some blacklist based methods. In such detection methods, the blacklists may be created and maintained through various techniques, such as manual labeling, honeyports, user feedbacks and crawlers. The blacklist based methods are simple and have a high accuracy. At the same time, they are incapable of detecting new, unseen malicious URLs, which often cause big harms to users.

The *content based methods* analyze the content of the corresponding web page of a URL to detect whether the URL is malicious. Web page content provides rich features for detection. For example, Provos *et al.* [14] detected malicious URLs using features from the content of the corresponding URLs, such as the presence of certain javascript and whether iFrames are out of place. Moshchuk *et al.* [11] used anti-spyware tools to analyze downloaded trojan executables in order to detect malicious URLs. The content based methods are useful for offline detection and analysis, but are not capable of online detection. For online detection, the content based methods often incur significant latency, because scanning and analyzing page content often costs much computation time and resource.

Most recently, the *URL based methods* use only the URL structures in detection, even without using any external information, such as WHOIS, blacklists or content analysis. McGrath and Gupta [10] analyzed the differences between normal URLs and phishing URLs in some features, such as the URL length, domain name length, number of dots in URLs. Such features can be used to construct a classifier for phishing URL detection. Yadav *et al.* [16] examined more features, such as the differences in bi-gram distribution of domain names between normal URLs and malicious ones. Their study confirmed that normal URLs and malicious ones indeed have distinguishable differences in the features extracted from URLs themselves alone. Their study, however, did not propose any classifier for malicious outlier detection. Ma *et al.* [8] applied machine learning methods to construct classifiers for malicious URL detection. They used two kinds of features. The *host-based features* are related to the host information such as IP addresses, WHOIS data, and the geographic properties of the URLs. Those features are highly valuable for classification, but they may cause non-trivial latency in detection. To this extent, their method is not completely URL based. The second group of features they used is the *lexical features*, which include numerical information, such as lengths of features, number of delimiters, and existence of tokens in the hostname and in the path of the URL. Le *et al.* [7] showed that using only the lexical features still can retain most of the performance in phishing URL detection. Kan and Thi [6] also conducted URL classification based on lexical features only, but their work tries to assign categories to normal URLs, such as news, business, and sports, instead of detecting malicious URLs from normal ones.

Our method proposed in this paper is URL based, and uses only the lexical features. Although Ma *et al.* [8] and Le *et al.* [7] also use lexical features to

build classifier to detect malicious URLs, they don't explicitly extract the human interpretable URL patterns which are highly desirable in real applications. The URL patterns are very valuable because they can help people anaylze the malicious activities, for example, they can be used to identify the malicious campaigns behind the malicious activities, and be combined with human's knowledge to prompt the detection performance.

Our method also differs from [8, 7] on how to select the lexical features. In the previous studies [8, 7], the lexical features are the tokens in the URL strings delimited by characters "/", "?", ".", "=", "-", and "_". Tokens extracted using those delimiters may not be effective. For example, the previous method may not be able to identify pattern "`*biz*main.php`" from malicious URL segments "`mybiz12832main1.php`" and "`godbiz32421main2.php`". To tackle the problem, we develop a method to extract lexical features automatically and dynamically. Our method is more general than the existing methods in lexical feature extraction, and can use more flexible and informative lexical features.

## 3 Extracting Lexical Features

A URL can be regarded as a sequence of **URL segments**, where a URL segment is a domain name, a directory name, or a file name. As discussed in Section 2, the previous studies [8, 7] often treat a segment as a token in mining patterns and extracting features for malicious URL detection. Simply treating each URL segment as a token, however, may limit the detection of malicious URLs. For example, suppose we have malicious URLs containing segments "`mybiz12832main1.php`" and "`lucybiz32421main2.php`". If we treat a segment in whole as a token, we cannot extract the meaningful common substrings "`biz`" and "`main`" in those two segments.

In this section, we discuss how to extract the common substrings as lexical features in URLs that may be generated by the same malicious program. Such patterns will be the building blocks later for malicious URL detection. We start from URL segments, and then extend to segment sequences and URLs.

### 3.1 URL Segment Patterns

A URL segment is a string of predefined characters, as specified in the URL specification (`http://www.w3.org/Addressing/URL/url-spec.txt`). We define a URL segment pattern as follows.

**Definition 1 (Segment pattern).** *A **URL segment pattern** (or a* **segment pattern** *for short) is a string $s = c_1 \cdots c_l$, where $c_i$ $(1 \leq i \leq l)$ is a normal character defined by the URL specification, or $c_i = *$ where $*$ is a wildcard meta-symbol. Denote by $|s| = l$ the length of the segment pattern, and by $s[i] = c_i$ the $i$-th character in the segment pattern. We constrain that for any $i$ $(1 \leq i < l)$, if $c_i = *$, then $c_{i+1} \neq *$.*

*For two URL segment patterns $s = c_1 \cdots c_l$ and $s' = c'_1 \cdots c'_m$, $s$ is said to* **cover** *$s'$, denoted by $s \sqsupseteq s'$, if there is a function $f : [1, m] \to [1, l]$ such that*

1. $f(j) \leq f(j+1)$ *for* $(1 \leq j < m)$*; and*
2. *for any* $i$ $(1 \leq i \leq l)$*, if* $c_i \neq *$*, then there exists a unique* $j$ $(1 \leq j \leq m)$ *such that* $f(j) = i$*, and* $c'_j = c_i$*.* ∎

Trivially, a URL segment itself is a segment pattern that contains no "*" symbols.

*Example 1 (URL segment pattern).* "`*biz*main*.php`" is a segment pattern. It covers URL segment "`godbiz32421main2.php`". Segment pattern "`abc*xyz`" does not cover "`xyzabc`", and "`abc*abc`" does not cover "`abc`". ∎

The cover relation on all possible segment patterns form a partial order.

*Property 1.* Let $\mathcal{S}$ be the set of all possible segment patterns. $\sqsupseteq$ is a partial order on $\mathcal{S}$.

*Proof.* (Reflexivity) $s \sqsupseteq s$ holds for any segment pattern $s$ by setting $f(i) = i$ for $1 \leq i \leq |s|$.

(Antisymmetry) Consider two segment patterns $s = c_1 \cdots c_l$ and $s' = c'_1 \cdots c'_m$. Suppose $s \sqsupseteq s'$ under function $f : [1, m] \to [1, l]$ and $s' \sqsupseteq s$ under function $f' : [1, l] \to [1, m]$. We show $c_i = c'_i$ where $1 \leq i \leq m$ by induction.

(The basis step) If $c_1 \neq *$, then there exists a unique $j_1$ $(1 \leq j_1 \leq m)$ such that $f(j_1) = 1$ and $c'_{j_1} = c_1$. If $j_1 > 1$, then $f(j_1 - 1) < f(j_1) = 1$, a contradiction to the assumption that $f : [1, m] \to [1, l]$. Thus, $c_1 = c'_1$.

If $c_1 = *$, we assume $c'_1 \neq *$. Then, there exists a unique $i_1$ $(1 \leq i_1 \leq l)$ such that $f'(i_1) = 1$ and $c_{i_1} = c'_1$. Since $c_1 = * \neq c'_1$, $i_1 > 1$. This leads to $f'(i_1 - 1) < f(i_1) = 1$, and a contradiction to the assumption that $f' : [1, l] \to [1 : m]$. Thus, $c'_1 = * = c_1$.

(The inductive step) Assume that $c_i = c'_i$ for $1 \leq i \leq k$ $(1 \leq k < m)$. We consider two cases. First, if $c_k \neq *$ and thus $c'_k \neq *$, then, using an argument similar to that in the basis step, we can show $c'_{k+1} = c_{k+1}$. Second, if $c_k = c'_k = *$, then $c_{k+1} \neq *$ and $c'_{k+1} \neq *$. There exists a unique $j_{k+1}$ $(k + 1 \leq j_{k+1} \leq m$ such that $f(j_{k+1}) = k + 1$ and $c'_{j_{k+1}} = c_{k+1}$. If $j_{k+1} > k + 1$, then $f(j_{k+1} - 1) < f(j_{k+1}) = k + 1$. This leads to a contradiction to the assumption that $c_i = c'_i$ for $1 \leq i \leq k$ and $c_k \neq *$. Thus, $j_{k+1} = k + 1$, and $c_{k+1} = c'_{k+1}$.

(Transitivity) Consider three URL segment patterns $s = c_1 \cdots c_l$, $s' = c'_1 \cdots c'_m$, and $s'' = c''_1 \cdots c''_n$. Suppose $s \sqsupseteq s'$ under function $f : [1, m] \to [1, l]$ and $s' \sqsupseteq s''$ under function $f' : [1, n] \to [1, m]$. We can construct a function $g : [1, n] \to [1, l]$ by $g = f \circ f'$. For any $1 \leq i < n$, since $f'(i) \leq f'(i + 1)$, $f(f'(i)) \leq f(f'(i+1))$, that is, $g(i) \leq g(i+1)$. Moreover, for any $j$ $(1 \leq j \leq l)$, if $c_j \neq *$, then there exists a unique $k$ $(1 \leq k \leq m)$ such that $c'_k = c_j$ and $f(k) = j$. Since $c'_k \neq *$, there exists a unique $i$ $(1 \leq i \leq n)$ such that $c''_i = c'_k = c_j$ and $f'(i) = k$. Thus, $f(f'(i)) = f(k) = j$. Under function $g$, $s \sqsupseteq s''$. ∎

**Definition 2 (Maximal segment patterns).** *Given a set of URL segments or URL segment patterns* $S = \{s_1, \ldots, s_n\}$*, a URL segment pattern* $s$ *covers* $S$*, denoted by* $s \sqsupseteq S$*, if for each* $s_i \in S$*,* $s \sqsupseteq s_i$*.*

*Segment pattern $s$ is called a* **maximal segment pattern** *with respect to $S$ if $s \sqsupseteq S$ and there exist no other segment pattern $s' \sqsupseteq S$ such that $s \sqsupseteq s'$ and $s \neq s'$. Denote by $MAX(S) = \{s | s$ is a maximal segment pattern with respect to $S\}$ the set of maximal segment patterns.* ■

*Example 2 (Maximal segment patterns).* Consider $S = \{\texttt{abcabc}, \texttt{abcaabc}\}$. Segment pattern "`abc*abc`" is maximal with respect to $S$. Although `ab*abc` $\sqsupseteq S$, it is not a maximal segment pattern, since `ab*abc` $\sqsupseteq$ `abc*abc` and `ab*abc` $\neq$ `abc*abc`.

Interestingly, given a set of segments, there may exist more than one maximal segment patterns. For example, "`abca*bc`" is another maximal segment pattern with respect to $S$. $MAX(S) = \{$`abc*abc`, `abca*bc`$\}$. ■

### 3.2 Finding Maximal Segment Patterns

Given a set $S$ of segments extracted from a set of malicious URLs, how can we compute $MAX(S)$? If $S$ contains only two segments, the problem is similar to the longest common subsequence (LCS) problem and the longest common substring problem [9, 1]. Following from the known results on those problems, we can use dynamic programming to compute the maximal segment patterns.

Let $D$ be a set of segment patterns, we define a function $\Omega(D) = \{s | s \in D, \nexists s' \in D, s \sqsupseteq s', s \neq s'\}$, which selects the maximal segment patterns from a set. Moreover, for a segment $s = c_1 \cdots c_l$, we write the prefix $s[1, i] = c_1 \cdots c_i$ for $1 \leq i \leq l$. Trivially, when $i > l$, $s[1, i] = s$.

We also write $s \diamond c$ a sequence where a character $c$ is appended to the end of $s$. Specifically, if a `*` is appended to a sequence ended by a `*`, only one `*` is kept at the end of the sequence. For example, `abc` $\diamond$ `*` = `abc*` and `abc*` $\diamond$ `*` = `abc*`. Moreover, for a set of segment patterns $D$, we write $D \diamond c = \{s \diamond c | s \in D\}$.

Given two URL segments $s$ and $s'$, denote by $MAX_{s,s'}(i, j)$ the set of maximal segment patterns in the set $\{s[1, i], s'[1, j]\}$. Apparently, $MAX_{s,s'}(|s|, |s'|) = MAX(\{s, s'\})$. We can compute the function using dynamic programming by

$$MAX_{s,s'}(i,j) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ \Omega(\{MAX_{s,s'}(i-1, j-1) \diamond s[i], & \\ MAX_{s,s'}(i-1, j) \diamond *, MAX_{s,s'}(i, j-1) \diamond *\}) & \text{if } s[i] = s'[j] \\ \Omega(\{MAX_{s,s'}(i-1, j), MAX_{s,s'}(i, j-1)\}) \diamond * & \text{if } s_i \neq s'_j \end{cases}$$

Consider two segment patterns $s$ and $s'$. Without loss of generality, assume $|s| \geq |s'|$. At each step of computing $MAX_{s,s'}(i, j)$, function $\Omega$ is called once, which takes time $O(|D|^2(i+j))$, where $D$ is the set of segment patterns where the maximal segment patterns are derived. Since $D$ is often small, at most 2 in our experiments, we treat $|D|$ as a constant. Thus, the complexity of the dynamic programming algorithm is $\sum_{1 \leq i \leq |s|, 1 \leq j \leq |s'|}(i + j) = O(|s|^2|s'|)$.

To compute the maximal segment patterns on a set of URL segments, we have the following result.

**Theorem 1.** *Let $S = \{s_1, \ldots, s_n\}$ $(n > 2)$ be a set of URL segments.*

$$MAX(S) = \Omega(\bigcup_{s \in MAX(S-\{s_n\})} MAX(\{s_n, s\}))$$

*Proof.* Consider a segment pattern $s' \in MAX(S)$. Apparently, $s' \sqsupseteq s_n$ and $s' \sqsupseteq S - \{s_n\}$. Thus, there exists at least one segment pattern $s'' \in \Omega(\bigcup_{s \in MAX(S-\{s_n\})} MAX(\{s_n, s\})$ such that $s' \sqsupseteq s''$. For each segment pattern $s'' \in \Omega(\bigcup_{s \in MAX(S-\{s_n\})} MAX(\{s_n, s\})$, $s'' \sqsupseteq S$. If $s' \neq s''$, then $s'$ is not a maximal segment pattern. This contradicts to the assumption $s' \in MAX(S)$. Therefore, $MAX(S) \subseteq \Omega(\bigcup_{s \in MAX(S-\{s_n\})} MAX(\{s_n, s\})$. Moreover, due to the $\Omega$ function, $MAX(S) = \Omega(\bigcup_{s \in MAX(S-\{s_n\})} MAX(\{s_n, s\})$. ∎

To avoid some segment patterns that are too general to be useful in malicious URL detection, such as "`*a*`", we constrain that the non-* substrings in a segment pattern must have at least 3 characters of numbers or letters.

### 3.3 URL Segment Sequential Patterns

We can treat a URL as a sequence of URL segments.

**Definition 3 (URL Segment sequential pattern).** *A **URL segment sequential pattern** (or* **sequential pattern** *for short) $u = \langle s_1, \ldots, s_l \rangle$ is a sequence of URL segment patterns. $|u| = l$ is the length of the sequential pattern.*

*For two sequential patterns $u = \langle s_1, \ldots, s_l \rangle$ and $u' = \langle s'_1, \ldots, s'_m \rangle$, $u$ is said to* **cover** *$u'$, denoted by $u \sqsupseteq u'$, if $m \geq l$ and there is a function $f : [1, l] \to [1, m]$ such that $s_i \sqsupseteq s'_{f(i)}$ $(1 \leq i \leq l)$, and $f(j) < f(j+1)$ $(1 \leq j < l)$.* ∎

Please note that, to keep our discussion simple, we do not have a wildcard meta-symbol at the sequential pattern level. When $u$ covers $u'$, the segment patterns in $u$ cover the segment patterns in a subsequence of $u'$ one by one.

*Example 3 (URL segment sequential pattern).* ⟨`20207db*.deanard.*`, `file4`, `get*.php`⟩ is a segment sequential pattern. It covers ⟨`20207db09.deanard.com`, `dir`, `file4`, `get2.php`⟩. Sequential pattern ⟨`20207db*`, `*deanard*`⟩ does not cover ⟨`20207db09.deanard.com`⟩. ∎

Similar to segment patterns, the cover relation on all possible sequential patterns form a partial order.

*Property 2.* Let $\mathcal{U}$ be the set of all possible URL segment sequential patterns. $\sqsupseteq$ is a partial order on $\mathcal{U}$.

*Proof.* (Reflexivity) $u \sqsupseteq u$ holds for any sequential pattern $u$ by setting $f(i) = i$ for $1 \leq i \leq |u|$.

(Antisymmetry) Consider two sequential patterns $u = \langle s_1, \ldots, s_l \rangle$ and $u' = \langle s'_1, \ldots, s'_m \rangle$. Suppose $u \sqsupseteq u'$ under function $f : [1, l] \to [1, m]$ and $u' \sqsupseteq u$ under function $f' : [1, m] \to [1, l]$. Apparantly, $|u| = |u'|$, otherwise the longer

sequential pattern cannot cover the shorter sequential pattern. Since $f(i) < f(i+1)$ and $|u| = |u'|$, for any $1 \leq i \leq l$, $f(i) = i$, $s_i \sqsupseteq s'_i$. In the same way, for any $1 \leq i \leq m$, $s'_i \sqsupseteq s_i$. By the antisymmetry in property 1, $s_i = s'_i$.

(Transitivity) Consider three sequential patterns $u = \langle s_1 \dots s_l \rangle$, $u' = \langle s'_1 \dots s'_m \rangle$, and $u'' = \langle s''_1 \dots s''_n \rangle$. Suppose $u \sqsupseteq u'$ under function $f : [1, l] \to [1, m]$ and $u' \sqsupseteq u''$ under function $f' : [1, m] \to [1, n]$. We can construct a function $g : [1, l] \to [1, n]$ by $g = f \circ f'$. For any $1 \leq i < l$, since $f(i) < f(i+1)$, $f'(f(i)) < f'(f(i+1))$, that is, $g(i) < g(i+1)$. Moreover, for any $1 \leq i \leq l$, $s_i \sqsupseteq s'_{f(i)}$, and for any $1 \leq f(i) \leq m$, $s'_{f(i)} \sqsupseteq s''_{f'(f(i))}$, by the transitivity in property 1, $s_i \sqsupseteq s''_{f'(f(i))}$. Under function $g$, $u \sqsupseteq u''$. ∎

We can also define maximal sequential patterns.

**Definition 4 (Maximal URL segment sequential pattern).** *Given a set of URL segment sequential patterns $U = \{u_1, \dots, u_n\}$, a URL segment sequential pattern $u$ is said to* ***cover*** *$U$, denoted by $u \sqsupseteq U$, if for each $u_i \in U$, $u \sqsupseteq u_i$*

*A URL segment sequential pattern $u$ is called a* **maximal URL segment sequential pattern** *(or* ***maximal sequential pattern*** *for short) with respect to $U$ if $u \sqsupseteq U$ and there exists no other sequential pattern $u' \sqsupseteq U$ such that $u \sqsupseteq u'$ and $u \neq u'$.* ∎

*Example 4 (Maximal sequential patterns).* Consider $U = \{\langle$`abcabc, index`$\rangle, \langle$`abcaabc,index`$\rangle\}$. Sequential pattern $\langle$`abc*abc,index`$\rangle$ is maximal with respect to $U$. Although $\langle$`ab*abc,index`$\rangle \sqsupseteq U$, it is not maximal, since $\langle$`ab*abc,index`$\rangle \sqsupseteq \langle$`abc*abc,index`$\rangle$.

Given a set of sequential patterns, there may exist more than one maximal sequential patterns. For example, $\langle$`abca*bc,index`$\rangle$ is another maximal sequential pattern with respect to $U$. ∎

Similar to mining maximal segment patterns, we can use dynamic programming to compute maximal sequential patterns. We denote by $MAX(U)$ the set of maximal sequential patterns with respect to a set of sequential patterns $U$.

Let $E$ be a set of sequential patterns, we define a function $\Omega(E) = \{u | u \in E, \nexists u' \in E, u \sqsupseteq u', u \neq u'\}$, which selects the maximal sequential patterns from a set. Moreover, for a sequential pattern $u = \langle s_1, \dots, s_l \rangle$, we write the prefix $u[1, i] = \langle s_1 \cdots s_i \rangle$ for $1 \leq i \leq l$. Trivially, when $i > l$, $u[1, i] = u$. We also write $u[i] = s_i$ and $u'[i] = s'_i$.

Given two URL sequential patterns $u$ and $u'$, denote by $seqMAX_{u,u'}(i, j)$ the set of maximal sequential patterns in the set $\{u[1, i], u'[1, j]\}$. Apparently, $seqMAX_{u,u'}(|u|, |u'|) = seqMAX(\{u, u'\})$. We can compute the function by

$$seqMAX_{u,u'}(i,j) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ \bigcup_{s \in MAX(u[i], u'[j])} (seqMAX_{u,u'}(i-1, j-1), s) & \text{if } MAX(u[i], u'[j]) \neq \emptyset \\ \Omega(\{seqMAX_{u,u'}(i-1, j), seqMAX_{u,u'}(i, j-1)\}) & \text{if } MAX(u[i], u'[j]) = \emptyset \end{cases}$$

To compute the maximal sequential pattern on a set of segment sequences, we have the following result.

**Theorem 2.** *Let $U = \{u_1, \ldots, u_n\}$ $(n > 2)$ be a set of sequential patterns. Then, $seqMAX(U) = seqMAX(seqMAX(U - \{u_n\}) \cup \{u_n\})$.*

*Proof.* Consider a sequential pattern $u' \in seqMAX(U)$. Apparently, $u' \sqsupseteq u_n$ and $u' \sqsupseteq U - \{u_n\}$. Thus, there exists at least one sequential pattern $u'' \in seqMAX(seqMAX(U - \{u_n\}) \cup \{u_n\})$ such that $u' \sqsupseteq u''$. For each segment pattern $u'' \in seqMAX(seqMAX(U - \{u_n\}) \cup \{u_n\})$, $u'' \sqsupseteq U$. If $u' \neq u''$, then $u'$ is not a maximal sequential pattern. This contradicts to the assumption $u' \in seqMAX(U)$. Therefore, $seqMAX(U) \subseteq seqMAX(seqMAX(U - \{u_n\}) \cup \{u_n\})$. Moreover, due to the $seqMAX$ function, $seqMAX(U) = seqMAX(seqMAX(U - \{u_n\}) \cup \{u_n\})$. ∎

### 3.4 URL Patterns

Although we can treat a URL as a sequence of URL segment patterns, due to the big differences among the roles of domain name, directories, and file name in a URL, we clearly distinguish those three parts in our definition of URL patterns.

**Definition 5 (URL Pattern).** *A* **URL pattern** *is a tuple $p = (h, d, f)$, where $h$ is a URL segment pattern corresponding to the domain name, $d = \langle s_1, \ldots, s_n \rangle$ is a URL sequential pattern corresponding to the directory path, and $f$ is a URL segment pattern represent the file name.*

*For two URL pattern $p = (h, d, f)$ and $p' = (h', d', f')$, $p$ is said to* ***cover*** *$p'$, denoted by $p \sqsupseteq p'$, if $h$ covers $h'$, $f$ covers $f'$, and $d$ covers $d'$.* ∎

**Definition 6 (Maximal URL pattern).** *Given a set of URL patterns $P = \{p_1, \ldots, p_n\}$, a URL pattern $p$ covers $P$, denoted by $p \sqsupseteq P$, if for each $p_i \in P$, $p \sqsupseteq p_i$.*

*A URL pattern $p$ is called a* **maximal URL pattern** *with respect to $P$ if $p \sqsupseteq P$ and there exists a URL pattern $p' \sqsupseteq P$ such that $p \sqsupseteq p'$ and $p \neq p'$. We denote by $urlMAX(P)$ the set of maximal URL patterns with respect to $P$.* ∎

Based on all the previous discussion and Theorems 1 and 2, we have the following result immediately.

**Theorem 3.** *Let $P = \{p_1, \ldots, p_n\}$ $(n > 1)$ be a set of URL patterns. $urlMAX(\{p_1, p_2\}) = \{(h, d, f) | h \in MAX(h_1, h_2), d \in seqMAX(d_1, d_2), f \in MAX(f_1, f_2)\}$. When $n > 2$, $urlMAX(P) = urlMAX(urlMAX(P - \{p_n\}) \cup p_n)$.* ∎

## 4 Mining Patterns

In Section 3, we discuss how to extract patterns as features from a set of URLs generated by a common malicious mechanism. Given a set of malicious URLs that are generated by different mechanisms, how can we extract meaningful patterns for detection? One fundamental challenge is that URL patterns are

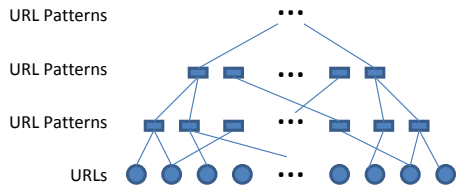



**Fig. 1.** Bottom-up search of URL patterns

generated from URLs, and cannot be assembled using any pre-defined elements, such as a given set of items in the conventional frequent pattern mining model. Thus, all existing frequent pattern mining methods cannot be used.

In this section, we develop new methods to mine URL patterns. We start with a baseline method that finds all URL patterns. Then, we present a heuristic method that finds some patterns that are practically effective and efficient.

### 4.1 The Complete Pattern Set Algorithm (CA)

As discussed, a URL pattern can be generated by a set of URLs. Given a set of URLs, the complete set of URL patterns are the maximal URL patterns from different subsets of the URLs. To search the complete set of URL patterns systematically, we can conduct a bottom-up search, as illustrated in Figure 1. We first compute the maximal URL patterns on every pair of malicious URLs, and add the generated valid URL patterns into a result pattern set. Then, we further compute the maximal patterns on the resulting URL patterns, and repeat this process until we can not generate new URL patterns. The pseudocode is shown in Algorithm 1, and the algorithm is called the **complete pattern set algorithm (CA)**.

Depending on different situations, the definition of "valid patterns" may be different. In our system, the patterns contain only "*" (empty) or only a full domain name are invalid, because the former are too general, and the latter appear only in one malicious domain, and are ineffective in detecting unseen URLs. Some other heuristics can also be applied. For example, pattern contains only a file type like ".jpg" should be removed.

We formally justify the correctness of the algorithm.

**Theorem 4.** *Algorithm 1 outputs the complete set of maximal URL patterns.*

*Proof.* Apparently, every pattern output by Algorithm 1 is a maximal pattern on some subsets of $P$. Let $p$ be a valid maximal URL pattern generated from a subset $P' \subseteq P$. We only need to show that Algorithm 1 outputs $p$. Since $p$ is valid, every patterns $p'$ such that $p \sqsupseteq p'$ must also be valid.

We show that a valid pattern generated from a subset of $l$ URLs will be included into $A$ in the algorithm no later than the $(l-1)$-th iteration by mathematical induction.

**Algorithm 1** Complete Pattern Set Algorithm (CA)

```
P ← malicous URLs set;                          ▷ P is the training URL pattern set
A ← ∅;                                          ▷ A is the result URL pattern set
repeat
    R ← ∅;                                      ▷ R is a intermediate URL pattern set
    for all pattern p ∈ P do
        for all pattern p′ ∈ P − {p} do
            pattern set Q ← urlMAX({p, p′});
            if Q is valid and Q ∉ A then
                R ← R ∪ Q, A ← A ∪ Q;
            end if
        end for
    end for
    P ← R;
until R = ∅;
return A;
```

(The basis step) The first iteration generates all valid patterns on every pair of URLs.

(The inductive step) Assume that all valid patterns generated by subsets of $k$ URLs are included into $P$ in the algorithm in no later than the $(k-1)$-th iteration. Consider a valid pattern $p$ that is generated by a subset of $(k+1)$ URLs $P' = \{r_1, \ldots, r_k, r_{k+1}\}$. Clearly, the patterns $p_1$ generated from $\{r_1, \ldots, r_k\}$ and $p_2$ generated from $\{r_2, \ldots, r_{k+1}\}$ are generated in the same iteration. If pattern $p$ has not been generated using $p_1$ and $p_2$ in a previous iteration, it is generated in the $k$-th iteration. ∎

The complexity of the CA algorithm is exponential to the number of input malicious URLs.

### 4.2 The Greedy Selection Algorithm (GA)

In real applications, the size of training data (malicious URLs) can be huge. The CA algorithm can be very costly. In the CA algorithm, we compare every malicious URL with all other URLs to extract all patterns. Consequently, one URL may be covered by many URL patterns. Although those patterns are different, they may heavily overlap with each other. In other words, there may be many redundant features in the result URL pattern set.

In our real data sets, the major features of a URL can be captured well after a small number of patterns are extracted. Moreover, one malicious URL generation mechanism is often represented by a non-trivial number of URLs in the training data set. Thus, we develop a greedy algorithm. In this algorithm, we join two URL patterns $p_1$ and $p_2$ only if the quality of the resulting pattern is better than that of $p_1$ and $p_2$. To avoid joining a URL many times, which may result in extracting too much redundant information, we remove a URL once

**Algorithm 2** Greedy Pattern Selection Algorithm

```
P ← malicous URLs set;                          ▷ P is the training URL pattern set
A ← ∅;                                          ▷ A is the result URL pattern set
repeat
    R ← ∅;                                      ▷ R is a intermediate URL pattern set
    for all pattern p ∈ P do
        for all pattern p′ ∈ P − {p} do
            pattern set Q ← urlMAX({p, p′});
            for all pattern q ∈ Q do
                if q is valid and q ∉ A and Sc(q) > Max(Sc(p), Sc(p′)) then
                                                ▷ Sc(q) is the quality of pattern q
                    R ← R ∪ {q}, A ← A ∪ {q};
                    P ← P − {p, p′};
                end if
            end for
        end for
    end for
    P ← R;
until R = ∅;
return A;
```

it generates a valid URL pattern. This algorithm is called the greedy selection algorithm (GA), as shown in Algorithm 2.

### 4.3 Indexing

The space of malicious URLs is very sparse. Most of the URLs do not share any common features. Thus, we can build an index on the URLs to facilitate similar the retrieval of similar URLs so that meaningful URL patterns can be mined.

In the system, we build an inverted index $IL_{file}$ on the trigrams of file names in malicious URLs. Similarly, we also build an inverted index $IL_{dir}$ on the trigrams of directory names, and an inverted index $IL_{domain}$ on the trigrams of domain names. We also build the corresponding inverted lists on white URL data, including $IL_{wfile}$, $IL_{wdir}$, and $IL_{wdomains}$. To avoid many meaningless trigrams such as "jpg", "com", "htm", and "www", we remove the top 2% most frequent trigrams of white data set from the malicious trigram inverted indexes. This step is similar to the well recognized stop-word removal in information retrieval. After building the indexes, when running our algorithms, we only need to compute the common pattern between those URLs sharing at least one trigram.

Consequently, the space complexity of the inverted index is linear to the malicious URLs. Since we do not need to store many patterns in main memory, the memory usage of our algorithm is very moderate.

### 4.4 Quality of URL Patterns

Many URL patterns are generated from a large training data set of malicious URLs. Thus, we need a method to assess the quality of the URL patterns and

select the effective ones. Based on the assumption that an effective pattern in malicious URL detection should appear frequently in different malicious domains and be infrequent in the white websites, we use two criteria: the malicious frequency and white frequency to estimate the quality of them. Please note that our white data set still may contain malicious URLs.

For a URL pattern $p$, the **malicious frequency** of $p$ is the number of unique domains it covers in the malicious training data, and the **white frequency** of $p$ is the number of unique domains it covers in the white data set.

## 5 Experimental Results

In this section, we report an extensive empirical study to evaluate our approach. The experiments are run on a PC with 2 dual-core 3.1 GHz processor with 8 GB memory, running Windows 7 operating system.

### 5.1 Data Sets

We use the real data from Fortinet. There are two URL data sources: the web-filtering URL feeds, and the URL log files. The web-filtering URL feeds are the labeled URLs from different anti-malicious service sources, for example, the Fortinet web-filtering rating engine [3], phishtank [13], and netcraft [12]. The URL log files are the URL records collected from different log files, such as the user browsing log files, the email URL log files, and the anti-spam log files. Since there are many unlabeled URLs in log files, they are usually the data where the detection system is to be applied. Because of the limited number of labeled URLs in log files, the detection models may first need to be trained from the web-filtering URL feeds, and then applied on the log files to detect new malicious URLs.

In addition, for online detection of malicious URLs, there may exist no labeled training URLs from the data sources where the detection will be applied. In such cases, the training data can first be collected from other data sources, such as malicious URLs made known in public domain, and train some initial patterns. These initial patterns can then be used to catch new suspicious URLs, and the verified malicious URLs can be added into the training data set to update the patterns. Thus, we evaluate the detection performance in two aspects: the performance when the training data and testing data are from the same data source, and the performance where the training and testing are conducted on different data sources.

For the datasets, we collect 0.5 million labeled malicious URLs and 1 million benign URLs from the web-filtering URL feeds. We also get two datasets from the URL log files: 35000 labeled malicious URLs and 70000 benign URLs. Table 1 summarizes some statistics of these four datasets.

**Table 1.** Some statistics of the data sets.

| | web-filtering URL feeds | | URL log files | |
|---|---|---|---|---|
| | malicious | benign | malicious | benign |
| number of URLs | 500K | 1000K | 35K | 70K |
| number of unique domain names | 279683 | 173859 | 9765 | 6367 |
| number of unique directory names | 73557 | 274105 | 5987 | 9807 |
| number of unique file names | 67460 | 389648 | 1760 | 9505 |
| average URL length | 79 | 132 | 77 | 116 |

### 5.2 Effectiveness of the GA Algorithm

We first run GA algorithm on the two data sets (0.5 million malicious URLs and 1 million benign URLs) from web-filtering URL feeds. Similar to previous works, we conduct 50/50 split cross-validation for training and testing. We also set different quality score threshold in the experiments to test the effectiveness of pattern quality score.

As described in Section 4, we use quality score to measure the quality of generated patterns. In the experiments, we implement the quality score as malicious probability ratio.

**Definition 7 (Malicious probability ratio).** *For a pattern p, the* **malicious probability** *of p is* $\frac{p\text{'s malicious frequency}}{n_b}$*, where* $n_b$ *is the number of unique malicious domains in the malicious data set. The* **white probability** *of p is* $\frac{p\text{'s white frequency}}{n_w}$*, where* $n_w$ *is the number of unique white domains in the white data set. If p's white frequency is 0, we assign the probability of it a small number. In our experiments, we use* $0.1/n_w$*.*

*The* **malicious probability ratio** *of p is* $\frac{p\text{'s malicious probability}}{p\text{'s white probability}}$*.* ■

Figure 2 shows the result. As the quality score threshold increases, the precision increases because some low-quality URL patterns are filtered out. At the same time, the recall decreases. Overall, when the threshold is 10, we can get satisfying accuracy (over 95%).

When running the GA algorithm, 4 GB memory is consumed because there are millions of lexical string and index structures need to be stored during the algorithm. There are totally 81623 valid patterns be generated. The left of Table 2 shows some of the top ranking patterns found by our algorithm. These top patterns capture rich features, including segment patterns of domains, full directory names, full file names, and file name segment patterns. The right of Table 3 lists some of the bottom ranking patterns. These patterns rank low because they are too general, thus can not distinguish malicious against normal URLs well.

### 5.3 Comparison between GA and CA

In this section, we compare the runtime and detection performance of the CA and GA algorithm.

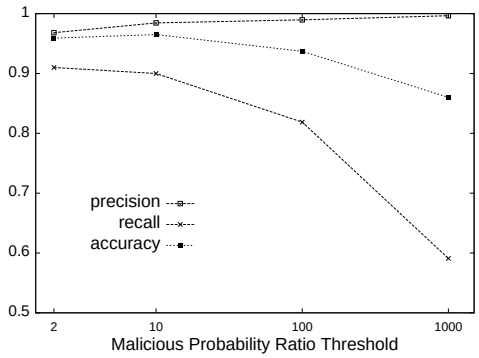



**Fig. 2.** Effectiveness of URL patterns

**Table 2.** Some top ranking and bottom ranking malicious URL patterns

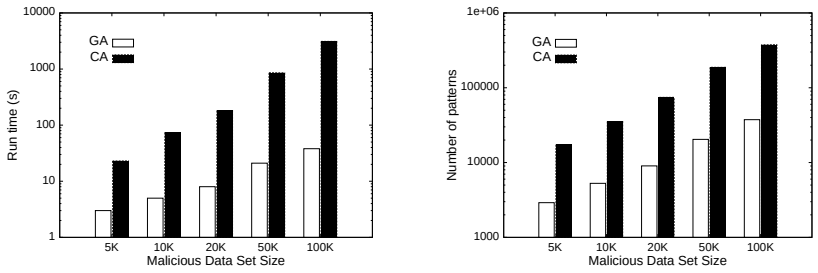
| Top Ranking Patterns |
|---|
| */*paypal*.com*/* |
| */IC/xvidsetup.exe |
| *.id3002-wellsfargo.com/index.php |
| *com.*natwest*/* |
| */account/paypal/* |
| */*cielo*/fidelidade.* |
| */vbv/*.html |
| */sais.php |
| ... |

| Bottom Ranking Patterns |
|---|
| ... |
| */Common/* |
| */2.0/* |
| */feeds/* |
| */blogs/* |
| */JS/* |
| */150/* |
| */br/* |
| */images/* |

Figure 3(a) compares the runtime of the CA and GA algorithms with respect to different training data size. Figure 3(b) shows the number of URL patterns found. Apparently, the CA algorithm takes much more time and generates much more URL patterns than the GA algorithm. How do the patterns generated by CA and GA, respectively, perform in detection?

In Figure 4, we compare the detection performance of the CA and GA algorithms by setting the training data set size to 200K. Although the CA algorithm generates much more URL patterns, the performance is not much better than the GA algorithm. The reason is that the patterns generated by the CA algorithm have heavy overlaps with each other. Many URL patterns actually point to the same feature, and most patterns cannot detect any malicious URLs.

In summary, the GA algorithm uses much less time and still captures important features from the training data.




(a) Run time (b) Number of patterns

**Fig. 3.** Compare between CA and GA algorithm in different training data size

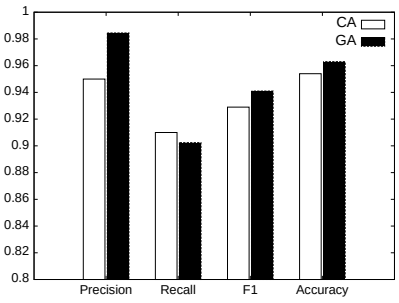


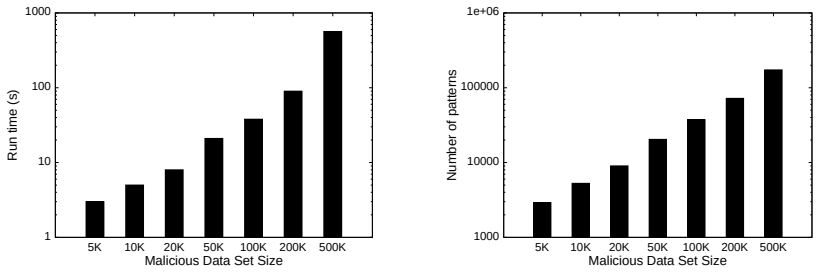

**Fig. 4.** Compare of GA and CA

### 5.4 Efficiency of GA

To verify the efficiency of the GA algorithm, we run it on malicious training data sets of different size. Figure 5 shows the runtime and the number of URL patterns generated with respect to the size of malicious training data set. Both the run time and the number of patterns grow quadratically when the training data set size increases. This is because in our algorithm, the patterns are generated by joining URL pairs. The run time is still acceptable even when the training data contains 500K malicious URLs and 1000K benign URLs.

### 5.5 Comparison with Some State-of-the-art Methods

Section 2 reviews the state-of-the-art machine learning based methods [8, 7] for classification on URLs. According to those studies, the confidence weighted algorithm (CW) [2] achieves the best results on URL lexical classification. Thus, we conduct experiments to compare our algorithm GA with the CW algorithm. We use the source code of the CW algorithm from Koby Crammer's webpage




(a) Runtime (b) Number of patterns

**Fig. 5.** Runtime and pattern number of GA on different training data

(`http://webee.technion.ac.il/people/koby/code-index.html`). We implement the lexical feature extraction as described in [8, 7]. Specifically, we treat each token (strings delimited by '/', '?', '.', '-', '=', and '_') in URLs as a binary feature.

We first evaluate the performance when the training data set and the testing data set are obtained from the same data source, which is the setting in [8, 7]. Please note that, in real applications, it is hard to collect reliable labeled training URLs from the data source where the detection system will be used.

We conduct a 50/50 split cross-validation over the 0.5 million malicious URLs and 1 million benign URLs from the web-filtering URL feeds data sets, respectively. Figure 6 shows the result, from the figure we can see that the GA algorithm has a higher precision, but the overall accuracy is a little lower than the CW algorithm due to a lower recall than that of the CW algorithm. The reason is that the GA algorithm extracts the patterns by joining URLs. Some patterns that appear only in one URL are not extracted. In the CW algorithm, even the infrequent lexical features are still used to get malicious weight.

To evaluate the performance of detection in a more practical setting, where the training data set and the testing data set are from different data sources, we repeat the experiments by applying the model trained from the web-filtering feeds data on the data sets of URL log files (35K malicious URLs and 70K benign URLs). The result is shown in Figure 7.

When the training data set and the testing data set are from different sources, the precision of the GA algorithm is much higher than that of the CW algorithm. When the training data set and the testing data set are from different sources, the CW algorithm may generate a biased model that leads to a low precision in the testing data set. In the GA algorithm, the strict quality score threshold can reduce the effect of noise and bias. In real applications, URL patterns can be further verified and modified by network security experts, which can further improve the detection accuracy.

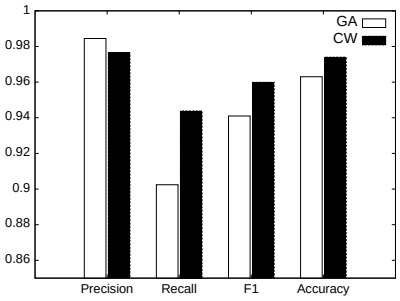



**Fig. 6.** Comparison with CW algorithm where training data and testing data are from the same source

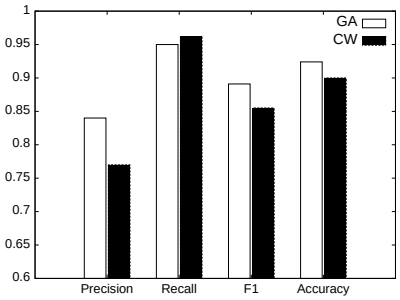



**Fig. 7.** Comparison with CW algorithm where training data and testing data are from the different source

In summary, comparing to the state-of-the-art methods, our algorithm achieves highly competitive performance.

## 6 Conclusions

In this paper, we tackle the problem of malicious URL detection and make two new contributions beyond the state-of-the-art methods. We propose to mine the human interpretable URL patterns from malicious data set and propose to dynamically extract lexical patterns from URLs, which can provide new flexibility and capability on capturing malicious URLs algorithmically generated by malicious programs. We develop a new method to mine our novel URL patterns, which are not assembled using any pre-defined items and thus cannot be mined using any existing frequent pattern mining methods. Our extensive empirical study using the real data sets from Fortinet clearly shows the effectiveness and

efficiency of our approach. The data sets are at least two orders of magnitudes larger than those reported in literature.

As future work, better methods to select patterns for detection is needed, because the malicious probability ratio threshold method may filter out not only noise patterns but also some useful patterns. How to further increase the recall of malicious detection is also critical. More lexical features beyond the common substring can be used in the detection, such as the statistics of segments, the meaning of segments (e.g., numerical or letter). Moreover, online algorithms can be developed to handle the evolving features over time.

## References

1. L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, pages 39–, Washington, DC, USA, 2000. IEEE Computer Society.
2. M. Dredze and K. Crammer. Confidence-weighted linear classification. In *In ICML 08: Proceedings of the 25th international conference on Machine learning*, pages 264–271. ACM, 2008.
3. Fortinet. Fortinet web filtering. `http://www.fortiguard.com/webfiltering/webfiltering.html`.
4. Gartner. Gartner survey shows phishing attacks escalated in 2007; more than $3 billion lost to these attacks. `http://www.gartner.com/it/page.jsp?id=565125`.
5. ICT Applications and Cybersecurity Division, Policies and Strategies Department, and ITU Telecommunication Development Sector. ITU study on the financial aspects of network security: Malware and spam. `http://www.itu.int/ITU-D/cyb/cybersecurity/docs/itu-study-financial-aspects-of-malware-and-spam.pdf`.
6. M.-Y. Kan and H. O. N. Thi. Fast webpage classification using url features. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, CIKM '05, pages 325–326, New York, NY, USA, 2005. ACM.
7. A. Le, A. Markopoulou, and M. Faloutsos. Phishdef: Url names say it all. In *Proceedings of the 30th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies*, pages 191–195, Shanghai, China, April 2011. IEEE.
8. J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Beyond blacklists: learning to detect malicious web sites from suspicious urls. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '09, pages 1245–1254, New York, NY, USA, 2009. ACM.
9. D. Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, Apr. 1978.
10. D. K. McGrath and M. Gupta. Behind phishing: an examination of phisher modi operandi. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, pages 4:1–4:8, Berkeley, CA, USA, 2008. USENIX Association.
11. A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware in the web. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'06)*, San Diego, California, USA, 2006. The Internet Society.
12. Netcraft. Netcraft. `http://news.netcraft.com/`.

13. PhishTank. Phishtank. `http://www.phishtank.com/`.
14. N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. In *Proceedings of the 17th conference on Security symposium*, pages 1–15, Berkeley, CA, USA, 2008. USENIX Association.
15. Wikipedia. Web threat. `http://en.wikipedia.org/wiki/Web_threat`.
16. S. Yadav, A. K. K. Reddy, A. N. Reddy, and S. Ranjan. Detecting algorithmically generated malicious domain names. In *Proceedings of the 10th annual conference on Internet measurement*, IMC '10, pages 48–61, New York, NY, USA, 2010. ACM.