CURLA: Cloud-Based Spam URL Analyzer for Very Large Datasets

Shams Zawoad, Ragib Hasan, Md Munirul Haque, and Gary Warner
{zawoad, ragib, mhaque, gar}@cis.uab.edu
Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35294, USA

***Abstract*—URL blacklisting is a widely used technique for blocking phishing websites. To prepare an effective blacklist, it is necessary to analyze possible threats and include the identified malicious sites in the blacklist. Spam emails are good source for acquiring suspected phishing websites. However, the number of URLs gathered from spam emails is quite large. Fetching and analyzing the content of this large number of websites are very expensive tasks given limited computing and storage resources. Moreover, a high percentage of URLs extracted from spam emails refer to the same website. Hence, preserving the contents of all the websites causes significant storage waste. To solve the problem of massive computing and storage resource requirements, we propose and develop CURLA – a Cloud-based spam URL Analyzer, built on top of Amazon Elastic Computer Cloud (EC2) and Amazon Simple Queue Service (SQS). CURLA allows processing large number of spam-based URLs in parallel, which reduces the cost of establishing equally capable local infrastructure. Our system builds a database of unique spam-based URLs and accumulates the content of these unique websites in a central repository, which can be later used for phishing or other counterfeit websites detection. We show the effectiveness of our proposed architecture using real-life spam-based URL data.**

***Index Terms*—Phishing, Cloud, Parallel Architecture, Spam URL**

I. Introduction

Phishing is one of the biggest ongoing threats to Internet users, who often become victim of this social engineering attack and lose their money to criminals. Phishing websites resemble legitimate websites, such as banks, product vendors, and service providers. Spam emails can contain URLs of these phishing websites. The goal of such spam emails is to draw users to visit phishing websites and deceive them to provide their private credentials, such as usernames and passwords, bank account numbers, and credit card numbers with pin codes on these fake websites. These secret credentials can then be used by malicious persons to withdraw money from the bank accounts and perform identity theft [1].

Though researchers have come across different approaches to detect and block phishing websites, phishing attack techniques continue to evolve. According to a recent fraud report by RSA, global losses from phishing are estimated to be at US$1.5 billion in 2012, which is a 22% increase from 2011 [2]. As mentioned in the Anti-Phishing Working Group (APWG) survey for the first half of 2012, there were at least 93,000 unique phishing attacks using approximately 64,000 unique domain names worldwide [3], which is an increase compared to the second half of 2011.

One way to decrease the rate of phishing is to improve the performance of phishing website detection systems, so that these websites can be blocked by Internet Service Providers (ISP) or browsers. However, to identify a phishing website, we need to access the source of possible phishing websites. The URL list gathered from spam emails is a big source of suspected phishing websites. However, the number of URLs coming from spam emails is very large. The University of Alabama at Birmingham (UAB) Phishing Data Mining Lab extracts nearly 1 million URLs daily from its spam email sources [4]. Analyzing this large number of URLs using the local computing resources is almost an impossible task, considering the time and storage it requires, and the corresponding expenses. Moreover, a good percentage of these URLs point to the same website, which introduces massive storage wastage if we fetch and store the content of the duplicate websites. A cloud-based parallel architecture can make it possible to analyze this large number of URLs coming from spam emails in reasonably short period of time, while reducing the local storage cost by sending only the new unique website's content to the local repository. Finally, by changing the IP of cloud instances frequently, we can avoid IP blacklisting by phishers.

To accomplish the goal, we propose CURLA, a cloud-based distributed and parallel content fetching and analyzing architecture that improves the performance of malicious websites detection and reduces the local storage cost by storing the contents of unique websites only. The spam database we used is located at UAB Phishing Data Mining Lab [4]. Whenever a new URL appears from spam emails, an Amazon Elastic Computing Cloud (EC2) [5] instance is assigned to fetch the content from the URL. After fetching the content, the instance generates MD5 hashes of the fetched files and compares with the previously stored hashes. Instead of checking the hash of one file with all the individual hash of previously stored websites' files, we propose a Bloom filter-based optimization technique, which improves the performance in terms of time and space significantly. When the matching algorithm finds a website as new, i.e. which has not been fetched before, the content of that website is forwarded to the local UAB storage. Otherwise, the website is marked as a duplicate copy of a previously found website. CURLA provides two levels of parallelism: machine level and thread level. While thread level parallelism is easily

achievable using local infrastructure, machine level parallelism using local infrastructure incurs massive cost and does not provide scalability. Using Amazon EC2 instances, however, we can scale the machine level parallelism to any desired level with little cost. Moreover, by changing the IP of the Amazon EC2 instances, we can protect the fetcher machines from being reverse blacklisted [6].

Contributions: The contributions of this paper are as follows:

1) We propose CURLA, a cloud-based distributed and parallel architecture to fetch and analyze large number of malicious websites' content in a reasonably short period of time.
2) CURLA removes the duplicate websites to rapidly identify malicious websites and reduces the local storage cost since we no longer need to store the content of duplicate websites.
3) We propose a Bloom filter-based website matching algorithm, which provides better space and time complexity compared to traditional matching procedures.
4) CURLA can be effectively scaled up to fetch and analyze large number of URLs by leveraging the scalability provided by cloud computing platforms. Based on workload, we can easily add more computing nodes with little configuration.

Organization The rest of the paper is organized as follows: Section II discusses the motivation behind this work. In Section III, we provide the architecture and workflow of CURLA. Section IV provides the experimental result, and Section V discusses the effects of utilizing CURLA on different aspects of counterfeit website detection. Section VI provides a review of related work and finally, we conclude in Section VII.

II. Research Motivation

A successful phishing attack consists of two parts: first, the creation of a fake website; and second, advertising the counterfeit website by the distribution of spam emails asking people to visit the site and share their login credentials. Today's spam emails are sufficiently personalized to deceive users and to make them follow the URL of phishing websites either by fear or greed. To prevent users from visiting known phishing websites, web browsers or ISPs often maintain a blacklist. To create the blacklist of phishing websites, we need to know which websites are real phishing websites and which are legitimate websites. URLs collected from spam emails is a great source of identifying potential phishing websites, since spam email is the most popular medium to spread the URLs of phishing websites. Besides blacklisting, security professionals at the victim organization, i.e. the one that is being imitated, must determine from a large collection of potential phishing sites which sites are actually phishing sites and are targeting their organization. A large collection of URLs can be extracted from spam emails targeting the victim, but then must be reviewed to find the phishing sites, which need to have an incident response action taken against them. However, the number of URLs collected from spam emails is very large, and examining and analyzing the URLs manually to determine whether or not a URL represents a legitimate website would be very difficult because of the amount of time, labor, and expense associated with it.

To overcome the problem of manual review, researchers proposed several content-based phishing website detection methods for automating the determination of phishing websites [7], [8], [9]. Content-based detection techniques generally download the content hosted at the URL and use features extracted from the content to check whether the website currently under review is sufficiently similar to an already confirmed phishing website. To use this approach, we need to fetch the content of all the websites found from spam emails. After that, we match the content with the previously confirmed phishing websites.

However, because of the large number of URLs coming from spam emails, there are two major problems when we want to analyze all the URLs in this manner. First, as fetching and analyzing a website's content require significant amount of time, the incident response time will be high if we cannot fetch and analyze large number of websites simultaneously. The response time is critical here because the success of phishing attacks depends on the longevity of phishing websites. Rapid action taken against the phishing websites can reduce the number of victims. Second, a significant portion of URLs point to the same website. Hence, a significant amount of computational and storage resources are wasted on these duplicated websites. For example, if the contents of one website requires 5MB and there are 200,000 duplicate instances of that website's URL, there will be 1TB of storage waste. Therefore, if we can populate a database of URLs that point to only unique websites and build a repository of those unique websites (prior to content-based phishing website identification), we can save a large amount of storage and identify a phishing website with greater efficiency.

To provide such a database and website repository service, we need to fetch and analyze all the URLs coming from spam emails and detect whether a URL points to a previously fetched website. Given the large number of URLs, we need a very costly local infrastructure to provide the desired functionality in a reasonable time period. Moreover, local infrastructures suffer from fixed public IP addresses, which can be blocked by malicious websites after a few attempts. As a cloud infrastructure is cost effective than a local infrastructure [10], using a cloud-based distributed architecture, we can achieve desired performance at a low cost. Additionally, we can change the public IP address of the cloud instances dynamically for not being reverse blacklisted [6].

III. Cloud-Based Spam URL Analyzer

The two main objectives of CURLA are to improve the performance of website analyzing by introducing parallelism, and to reduce the local storage cost by storing the contents of unique websites only. In CURLA, we used a message passing-based distributed and parallel architecture to fetch and analyze a large number of websites simultaneously using

Amazon EC2 instances. CURLA also reduces the storage cost by distributing the storage of duplicate websites among multiple cloud machines. In this section, we provide the architecture and work flow of CURLA.

A. Architecture

Figure 1 depicts the overview of the system and each of the modules are described in detail below:

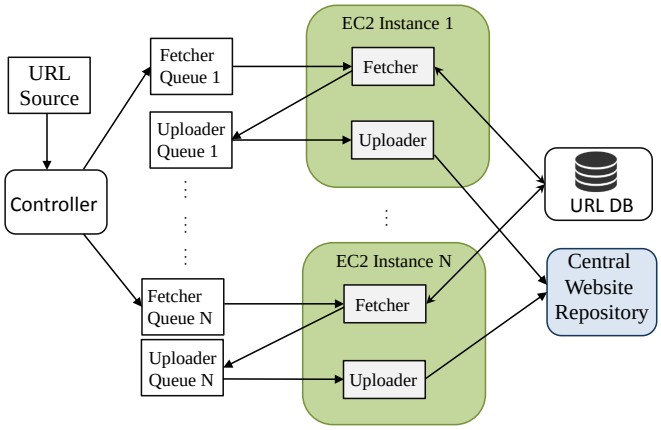


Fig. 1: CURLA System Architecture

Controller: The controller module triggers the total operation of the system. Controller gets URLs from a source of URLs. This URL source is developed by extracting URLs from spam emails. Controller is responsible to manage the fetcher and uploader message queues. It can also be enhanced to manage the Amazon EC2 instances. Controller has a list of fetcher queues. After receiving an URL from the URL source, the controller sends a URL-fetch-request message to a fetcher queue selected by round-robin fashion.

Fetcher Message Queue: Each of the fetcher message queues is an instance of Amazon Simple Queuing Service (SQS) [11]. Based on the number of fetcher instances, number of fetcher message queues can vary, where one queue is dedicated for one fetcher instance. It stores the URL-fetch-request messages received from the controller and dispatches to fetcher upon request. The queuing functionality is provided by the Amazon SQS API.

Fetcher: The fetcher module runs on Amazon EC2 instance. It is connected with the controller module through the fetcher message queue. Fetcher module receives a URL-fetch-request message from the fetcher message queue, then fetches the website content from the URL, and analyzes to check whether the website is new or duplicate copy of any previously fetched website. This module is capable of handling multiple URL-fetch-requests simultaneously. The task of fetching and analyzing one website is encapsulated into one thread and after receiving a message from the fetcher message queue, the fetcher spawns a thread. In this way, we can achieve thread level parallelism. Moreover, using multiple fetcher queues and fetcher instances, we can execute the fetching and analyzing task parallely in multiple machines, which gives us machine level parallelism.

Uploader Message Queue: Uploader Queues are also instances of Amazon SQS. Each uploader queue is attached with one fetcher module and one uploader module. It stores the website-upload-request messages received from the fetcher and dispatches to the uploader upon request.

Uploader: The uploader module is a FTP client running on Amazon EC2 instances and is responsible to upload the local copy of a fetched website to the central website repository. It receives a website-upload-request message from the uploader queue and uploads the website content to the central website repository using FTP. Each uploader module is connected with one fetcher module through an uploader message queue. One connected fetcher and uploader should run in the same Amazon EC2 instance to resolve the location of the local copy. This module is capable of handling multiple website-upload-request messages simultaneously. One uploader can spawn multiple threads for multiple upload requests. Additionally, multiple uploader instances connected with multiple uploader message queues provide machine level parallelism for website uploading.

URL Database: URL database contains information of all the unique and duplicate URLs. Using the hash of website files, we derived a Bloom filter-based algorithm to match the uniqueness of two websites. Hence, for each unique URL, we store only one Bloom filter [12] to preserve the hash information of all the downloaded files of a website.

Central Website Repository: This module stores the content of all the unique websites. An FTP server is running on this module to accept FTP connection from the uploader modules.

B. Work Flow

After receiving a new URL from the URL source, the controller sends that URL to a fetcher message queue. Once a fetcher instance receives this message, it starts the task of fetching and analyzing for the received URL. If the website is found as new, a website-upload request message is dispatched from a fetcher to an uploader using an uploader message queue. After receiving a website-upload request message, the uploader uploads the website content to the central website repository. The work flow of CURLA is depicted in Figure 2.

The process of getting a new URL from the URL source and sending it to a fetcher message queue is a straightforward task. This process can be executed from any local machine. The URL-fetch-request message contains only the URL path. One fetcher module is attached with one fetcher queue. The fetcher starts working after having a URL-fetch-request message in its fetcher message queue. Hence, the first component of the fetcher module is a message queue listener. The listener always searches for new messages in its associated fetcher queue. Whenever there is a new message found by the listener, it spawns a *fetcher-analyzer* thread.

The fetcher-analyzer thread has two functionalities. It first fetches the content of the website represented by a given URL, and analyzes whether the fetched website is new or a duplicate copy of any previously found website. Fetching is performed using an automated web crawler that uses GNUs wget [13]

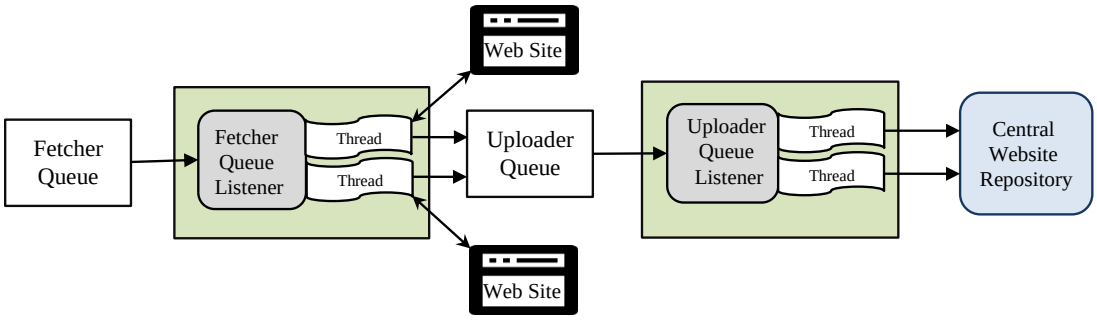


Fig. 2: Work Flow of CURLA

with one level of recursive fetching, time out, and ignoring robots options.

A naive approach of analyzing a website's content for finding duplicate website is to generate the hash of all the files (e.g., html, css, javascript, images, etc.) of the website and match the file-hashes with all the file-hashes of each of the websites found previously. If the number of files for the new website is *m*, the number of previously found website is *n*, and the average number of files for the previously found websites is *p*, then the complexity of the naive approach will be $O(mnp)$. To use this approach, we also need to store the hash of all files found for each of the websites. Hence, the storage requirement is $O(np)$. But our proposed Bloom filter-based matching algorithm reduces the time complexity to $O(mn)$ and storage complexity to $O(n)$.

A Bloom filter is a probabilistic data structure with no false negatives, which is used to check whether an element is a member of a set or not [12]. Bloom filter stores the membership information in a bit array. Element insertion time and membership checking can be done in constant time. The only drawback of the Bloom filter is the probability of finding false positives. However, we can reduce the probability of false positives by using a larger bit array.

To develop a Bloom filter-based algorithm, we use one Bloom filter to store the membership information of all the files of one website. Hence, for *n* number of websites, there will be *n* number of Bloom filters. When a website is found as new, we first create an empty Bloom filter for the new website. To insert the membership information of a file of that website into the Bloom filter, we first create a MD5 hash of the file (referred as file-hash). After that, we generate the *k* number of bit positions for the file-hash by hashing the file-hash for *k* times, and update those *k* bit positions of the Bloom filter. In this way, we can preserve membership information of large number of files in a single Bloom filter. Since we do not need to store the hash of each files of a website, the storage cost reduces from $O(np)$ to $O(n)$.

At the time of matching the content of a newly fetched website, the analyzer selects a previously found website and its Bloom filter. Then, for each of the files of the new website, it creates a MD5 hash of the file and checks whether it exists in the selected Bloom filter or not. The membership information checking can be done in constant time. If a certain percentage of files (we used 90%) of the new website exists in the selected Bloom filter, we then say that the new website is a duplicate copy of the selected website. Otherwise, the new website will be treated as a unique website. For *m* number of files of the new website, this operation can take maximum *O(m)* time. Hence, for *n* number of previously stored websites, the worst case time complexity will be $O(mn)$. Algorithm 1 describes the matching procedure in detail.

Algorithm 1 Website Matching Using Bloom Filter

```
1:  bloomFilter ←Bloom filter with 0.1% false possitive
2:  probability for 100 elements
3:  fileList ← List < File >
4:  fileHashList ← List < String >
5:  urlBloomList ← get bloom filter content for all the
6:  previously found websites
7:  for all file in fileList do
8:      insert MD5(file) to fileHashList
9:  end for
10: for all urlBloom in urlBloomList do
11:     clear bloomFilter
12:     set content of urlBloom in bloomFilter
13:     matchCount ← 0
14:     for all fileHash in fileHashList do
15:         if bloomFilter contains fileHash then
16:             matchCount ← matchCount + 1
17:         end if
18:     end for
19:     if matchCount/size of fileHashList >= desired matching percentage
        then
20:         matched successfully and return
21:     else
22:         not matched
23:     end if
24: end for
```

After analyzing the content of a website, if the website is found to be new, the analyzer saves the URL along with its Bloom filter content in the unique URL table of the URL database and issues a upload-request message to the uploader message queue. The upload request message consists of the new URL and the location of the fetched content. On the other hand, if a website is found as a duplicate copy of a previously found website, we then store this URL in a duplicate URL table with the matched website's identification. In this case, the fetcher does not issue any upload-request message.

The Uploader module has a message queue listener, which listens for any new upload-request message and initiates a thread to handle the upload-request. Task of an uploader thread is to communicate with the central website repository through

FTP and transfer the local copy of a website to the central repository.

IV. Experiment and Evaluation

In this section, we provide an overview of the working data set, experimental setup, and evaluation of CURLA.

A. Data Set

The dataset for this research is from the UAB Phishing Data Mine [4]. Table I presents number of spam emails, total no. of URLs, and no. of URLs that have unique full path gathered between Nov 7 to Nov 11, 2013. We can reduce the number of URLs to a great extent by checking the uniqueness of URL path. Even after this reduction, number of unique URLs is more than 200,000 on each day. A good percentage of these unique URLs are redundant that we can only identify after analyzing the content of the websites.

Date	No. of Spams	No. of URLs	No. of Distinct URLs
Nov 7	909,471	1,419,667	427,185
Nov 8	959,124	1,285,033	272,487
Nov 9	759,293	958,076	242,210
Nov 10	859,277	874,654	202,696
Nov 11	946,668	1,026,624	274,226

TABLE I: URL Statistics

B. Experimental Setup

System Configuration: In our experiments, we used a Dell laptop running Debian 3.2.46-1 on Intel Core 2 Duo CPU (2.66GHz) with 4GB of RAM as the controller. The controller program is built on OpenJDK (version:1.6.0_27). We used AWS SDK for Java [14] to manage the fetcher and uploader message queues.

We used four Amazon EC2 small instances (m1.small) to fetch, analyze and upload the unique content to the central repository. Each of the instances running Ubuntu 12.04.2 LTS operating system and as reported by the OS, the CPU is single core Intel(R) Xeon(R) CPU E5-2650 (2.00GHz) with 1.6GB RAM. The fetcher and uploader running on these instances are built on OpenJDK(version:1.6.0_27).

We used one Amazon EC2 large instance (m1.large) as a central repository of the unique websites and URL database. The OS of this instance is Ubuntu 12.04.2 LTS, and the CPU has dual Intel(R) Xeon(R) E5507 (2.27GHz) processors. A FTP server was installed in this machine to upload the website content through FTP. The URL database was maintained by PostgreSQL 9.1.9.

C. Evaluation

In our experiments, we did not use all the URLs, rather we used 7600 unique URLs of Nov 11th divided in five groups, each contains 100, 500, 1000, 2000, and 4000 URLs respectively and no URL is common between two groups. While referring to individual date, we choose Nov 11, which refers to most recent data. Table II presents some statistics of URL analyzing results. As we notice from Table II, for a large number of URLs, we failed to fetch the content. A possible reason is that the websites went down before fetching. We ignore the robots.txt file in wget, hence, the wget is not blocked because of the robots.txt. For a small percentage of URLs (average 4%), we could not determine their status.

No. of Test URLs	No. of Unique Websites	No. of Duplicate Websites	No. of Websites Failed to Fetch	% of Websites Successfully Analyzed
100	40	11	43	94.00
500	130	126	237	98.60
1000	224	283	457	96.40
2000	460	604	837	95.05
4000	855	1287	1701	96.08

TABLE II: URL Analyzing Results

Storage Savings: As we notice from Table II, a large number of URLs point to same website, we can save a significant amount of storage of local repository by ignoring the duplicate content. Figure 3a presents the amount of duplicate storage that we can save while analyzing a set of unique URLs. The best fit function that matches with the experimental result is:

$$Storage\ Savings = 2975 * No.\ of\ URL(Bytes) \quad (1)$$

From this function, we can interpolate the amount of storage that we can save in each day or in a given time period. According to equation 1, on Nov 11 2013, we can save $2975 * 274226 = 778$ Mega Bytes of storage. Considering the number of such URLs easily reaching to billions and their ever increasing nature, the storage saving can easily reach in units of Terra Bytes.

We can also estimate the amount of storage savings from the ratio of storage required for duplicate websites to the unique websites. Figure 3b represents the ratio of duplicate vs. unique storage with the increase in number of URL. The best fit function which matches with the experimental result is:

$$\%\ of\ Duplicate\ Storage = 0.0019 * No.\ of\ URL \quad (2)$$

According to equation 2, on Nov 11, storage required for duplicate websites is 5 times higher than the storage required for unique websites.

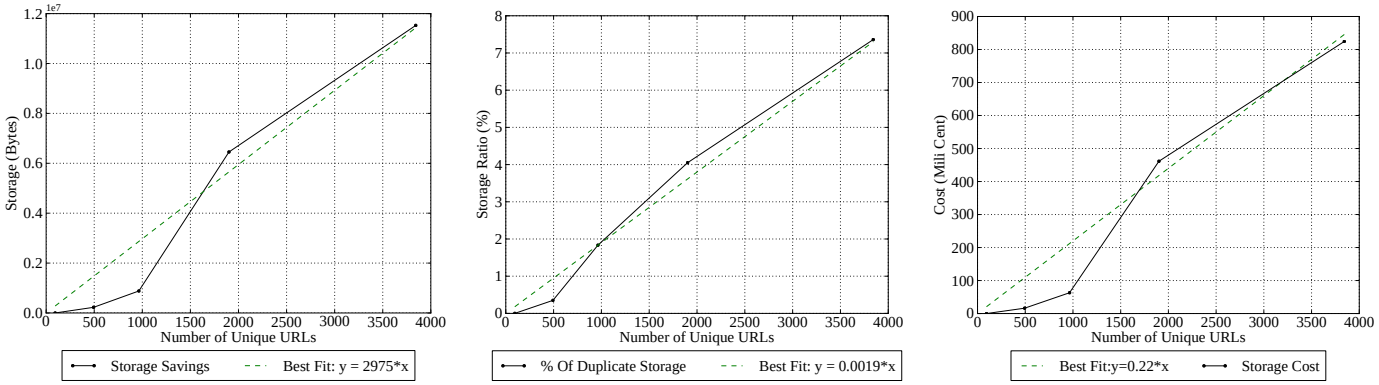
From the amount of storage that we can save in the local infrastructure, we can determine how much this storage would cost if we had preserved the duplicate content. Dutta et al. proposed a full cost accounting model to identify the cost of one byte and according to their case study, cost per byte for the data center of Computer and Information Sciences department of UAB is $71.51X10^3$ pico cents [15]. We used this value to identify the cost that we can save due to duplicate storage, which is illustrated in Figure 3c. To interpolate the amount of cost in a single day, we identify the following best fit function:

$$Cost = 0.22 * No.\ of\ URL(mili\ cent) \quad (3)$$

According to Equation 3, for the 274,226 unique URLs of Nov 11, we can save approximately $0.60.

Table III represents the estimated amount of duplicate storage, ratio of duplicate storage and unique storage, and cost incurred for the duplicate storage for the five days of data.

Estimated Number of Instances: We can estimate the number of required Amazon instances based on some assumptions.



(a) Storage Required for Duplicate Websites (b) Duplicate vs. Unique Storage Ratio (c) Cost Analysis

Fig. 3: Storage Savings Analysis

Date	No. of Distinct URLs	Duplicate Storage (GB)	Duplicate vs. Unique Storage (%)	Duplicate Storage Cost (Cents)
Nov 7	427,185	1.18	811.65	93.98
Nov 8	272,487	0.75	517.72	59.95
Nov 9	242,210	0.67	460.2	53.29
Nov 10	202,696	0.56	385.12	44.59
Nov 11	274,226	0.75	521.03	60.33

TABLE III: Interpolated Results of Storage Savings

We assume that time required to fetch a website by small (m1.small) instance is t_f seconds, time to analyze is t_a, and time to send the upload request is t_r. Hence, processing time for one URL by one m1.small instance is (t_f + t_a + t_r) seconds. If we want to process U number of URLs and allow H number of threads for each instance, then the total number of instances (N) required to process all the URLs in T seconds can be defined by following equation:

$$N = \frac{U(t_f + t_a + t_r)}{T * H} \quad (4)$$

For a realistic assumption of the required number of instances, we measured t_f, t_a, t_r, which is represented in figure 4. As a lower bound, we can select the average time of each step, and for higher bound, we can select the sum of average and standard deviation time. According to Equation 4, the lower bound for estimated number of instances (N_{lower}) to process 10,000 URLs in 50 minutes, with maximum 10 threads is:

$$N_{lower} = \frac{10000(t_{f,avg}+t_{a,avg}+t_{r,avg})}{50*60*10}$$

$$N_{lower} = \frac{10000(8.7+0.19+0.47)}{50*60*10} = 3 \text{ (Approximately)}$$

The corresponding upper bound is:

$$N_{upper} = \frac{10000(323.91+3.81+5.98)}{50*60*10} = 111 \text{ (Approximately)}$$

Estimated Cost: Based on the estimated number of instances, we measured the estimated cost to run the system using small (m1.small) instances. Cost of running a small instance for one hour is \$0.06 [16]. According to Equation 4, we can determine estimated number of instances to process 274,226 URLs of Nov 11, 2013 for different completion time, with maximum 10 threads and can calculate the required cost, which is represented in Table IV.

Targeted Time(Hour)	N_{upper}	N_{lower}	$Cost_{upper}$(\$)	$Cost_{lower}$(\$)
1	2542	71	152.52	4.26
2	1271	35	152.52	4.20
3	847	24	152.46	4.32
4	635	18	152.40	4.32
5	508	14	152.40	4.20
6	423	12	152.28	4.32
7	363	10	152.46	4.20

TABLE IV: Estimated Cost of Using Amazon EC2

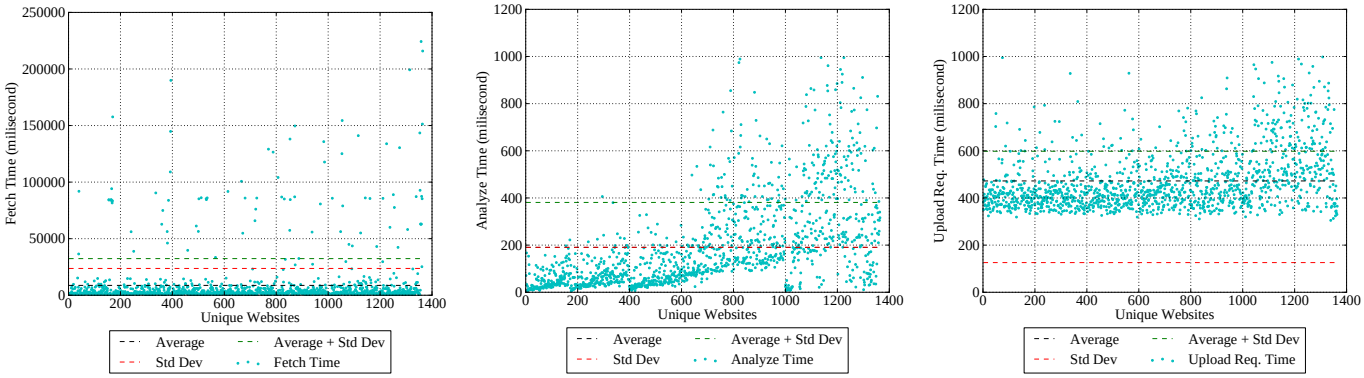
However, the cost represented in table IV will be nearly six times less than the existing cost if we go for three years contract. There are other costs for Amazon SQS for the queuing service, which is not significant for one day (\$0.50 per 1 million Amazon SQS Requests [17]). Data in and out form Amazon EC2 instance is also not very significant (Up to 10 TB / month 0.12USD per GB [16]).

Bloom Filter-based Matching Algorithm: Performance improvement of Bloom filter-based matching algorithm is depicted in Figure 5, which provides the analyzing time for one new website with the increase in number of previously found websites. For better measurement of time, we ran the experiment in offline mode using a single threaded program. For the 2,000 URLs of experiment, difference between our proposed approach and naive approach is nearly 1 seconds, however, for the URLs of Nov 11 the difference is 2.34 minutes.

V. Discussion

In this section, we discuss the effect of CURLA for phishing website detection and applicability of CURLA in other areas.

Cost vs. Performance: As we notice from Table IV, the cost of using Amazon EC2 instances does not decrease with lower performance. Hence, we can scale up the distributed infrastructure according to our desired performance without any extra cost. For average case, we can analyze all the URLs



(a) Fetch Time (b) Analyze Time (c) Upload Request Time

Fig. 4: Time Required in Different Steps

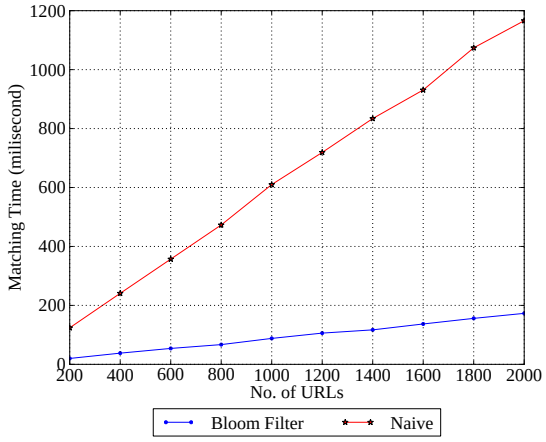


Fig. 5: Performance Analysis of Matching Algorithm

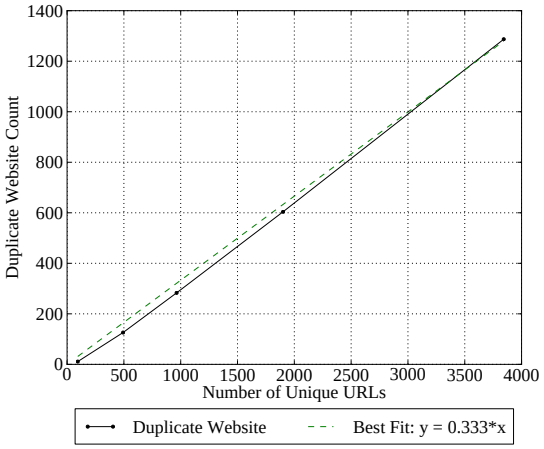


Fig. 6: Growth Rate of Duplicate Websites

of Nov 11 2013 in one hour with approximately $4. However, to achieve the similar performance using local infrastructure, we need 71 computers. Developing such a local infrastructure requires a great deal of investment and maintenance cost. We also notice that on Nov 7th, there are almost twice the amount of URLs compared to other 4 days. It is difficult to scale up the local infrastructure when there will be more URL, or scale down when there will be less URL, which is easily achievable using cloud infrastructure. On the other hand, using a small fixed local infrastructure, for example using 10 computers, the whole task will require nearly 7 hours for the URLs of Nov 11. For phishing detection, the response time should be as low as possible to minimize the effect of phishing and CURLA helps to minimize the phishing detection time in least cost.

Effect on the Performance of Phishing Website Detection: Besides saving the storage for duplicate websites, identifying the unique websites has effect on the performance of phishing website detection. Figure 6 represents the growth of number of duplicate websites with the increase in number of URLs. The best fit function that we found for experimental data set:

$$No.\ of\ Duplicate\ Websites = 0.333 * No.\ of\ URL \quad (5)$$

Based on Equation 5, we can project that there will be approximately 91,318 duplicate websites among the 274,226 URLs of Nov 11. Using the data provided by CURLA, a content-based phishing detection task does not look into these duplicate websites. If one phishing detection task requires 1 second, then we can save nearly 25 hours of computing time for the 91,318 duplicate websites.

The proposed scheme can also help to increase the phishing identification rate. Current approach of acquiring suspected phishing URLs from spam emails is analyzing the spam emails to determine whether the spam contains a phishing URL. This includes: searching for known brands, identifying pattern of email subject, etc. However, the phishers are also improving in writing the content of spam emails, or frequently changing the structure of the spam emails to spoof this technique. Hence, there is a chance to miss a possible phishing website using this technique. The only way to detect this is to fetch and analyze the content of the website represented by the URL. Hence, using the data gathered by CURLA, we can improve the rate of phishing website identification.

Other Application Area: Besides phishing websites, CURLA can be effectively used in identifying other types of counterfeit websites, which are also advertised by spam emails. For example, a significant portion of websites, we fetched are counterfeit websites for goods. These websites sell branded goods, which they are not authorized to sell. The organization

that owns the original brand needs to know which counterfeit websites sell their products to take necessary legal steps against those websites. Another example is illegal business websites, e.g., fake insurance companies, or business companies that send illegal spam campaign. To detect these illegal websites, we can fetch and analyze the content of such websites.

VI. Related Work

Content based phishing website detection has been explored by researchers and there are various established approaches [7], [18], [19], [20]. Content-based detection can combine techniques that draw features from the text of the main index page, characteristics of sets of component files, and measures of visual similarity among websites to identify phishing attacks. However, none of these solutions use cloud-based content fetching approach. Li et al. proposed a cloud-based phishing detection system named LARX that uses network traffic data archived at a vantage point and analyzes these data for phishing attacks [21]. All of LARX's phishing filtering operations use cloud computing platforms and work in parallel to handle large volume of data in a reasonable time.

The closest work related to our work is conducted by Ferguson et al. [6], where they proposed a cloud-based content fetching solution for phishing website analysis. Their main focus is to conceal anti-phishing probes from being detected and reverse blacklisted by the phishing websites. They used multiple cloud providers, such as Amazon EC2, Rackspace, and GoGrid, to initiate cloud instances and fetch website content. The purpose of the cloud-based clients is to fetch website content and send any data back to the controlling server to be stored. The controlling server serves as bridge between phishing database and cloud-based clients. We use the concept of concealing cloud-based website fetcher as stated in [6]. However, our system focuses on fetching the content of URLs coming from spam emails and finds the URLs with unique content that accomplishes different goals from [6].

VII. Conclusion and Future Work

In this paper, we have presented CURLA – a highly scalable and distributed cloud-based framework for analyzing the content of malicious websites. Deploying and maintaining a local infrastructure equivalent to CURLA will be cost inefficient. As CURLA only fetches and stores new and unique content, it can greatly reduce the cost of malicious website analysis in terms of time, storage, and money. Our experimental result shows the effectiveness of CURLA on small datasets and we project a large amount of savings in terms of storage, time, and cost for large scale datasets. The Bloom filter based website matching algorithm can also be blended with any other content-based counterfeit website detection methodologies. Utilizing our proposed system can also help to improve the rate of phishing and other types of counterfeit websites detection.

In future, we are planning to run CURLA for larger datasets and use different statistical methods to evaluate whether the best fit functions derived from the small datasets are also suitable for larger datasets. Moreover, a website that we detect as unique by comparing with the websites found in a single day may not be unique if we also consider the websites found in previous days. Hence, we plan to evaluate the performance of CURLA in detecting duplicate websites over a longitudinal history of data. Besides these, we plan to do a comprehensive cost analysis between local infrastructures and cloud-based infrastructures for large-scale spam url datasets.

Acknowledgments

This research was supported by NSF CAREER Award CNS-1351038, NSF Award, a Google Faculty Research Award, and the DHS Grant #FA8750-12-2-0254.

References

[1] M. Jakobsson and S. Myers, *Phishing and countermeasures: understanding the increasing problem of electronic identity theft*. WILEY, 2006.

[2] EMC, "The year in phishing," http://goo.gl/ounWgB, [Accessed Nov 4th, 2013].

[3] APWG, "Global phishing survey: Trends and domain name use in 1H2012," http://goo.gl/z0sbEB, [Accessed Nov 11th, 2013].

[4] The University of Alabama at Birmingham, "UAB phishing data mine," http://www.cis.uab.edu/UABSpamDataMine/, [Accessed Nov 11th, 2013].

[5] Amazon EC2, "Amazon elastic compute cloud (amazon ec2)," http://aws.amazon.com/ec2/, [Accessed Nov 11th, 2013].

[6] E. Ferguson, J. Weber, and R. Hasan, "Cloud based content fetching: Using cloud infrastructure to obfuscate phishing scam analysis," in *Proceedings of 8th World Congress on Services (SERVICES)*. IEEE, 2012, pp. 255–261.

[7] B. Wardman and G. Warner, "Automating phishing website identification through deep MD5 matching," in *Proceedings of the eCrime Researchers Summit, 2008*. IEEE, 2008, pp. 1–7.

[8] B. Wardman, T. Stallings, G. Warner, and A. Skjellum, "High-performance content-based phishing attack detection," in *Proceedings of the eCrime Researchers Summit, 2011*. IEEE, 2011, pp. 1–9.

[9] C. Whittaker, B. Ryner, and M. Nazif, "Large-scale automatic classification of phishing pages." in *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010.

[10] A. Khajeh-Hosseini, D. Greenwood, and I. Sommerville, "Cloud migration: A case study of migrating an enterprise it system to iaas," in *proceedings of the 3rd International Conference on Cloud Computing (CLOUD)*. IEEE, 2010, pp. 450–457.

[11] Amazon, "Amazon simple queue service (Amazon SQS)," http://aws.amazon.com/sqs/, [Accessed Nov 11, 2013].

[12] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[13] GNU wget, "Gnu project free software foundation(FSF)," http://www.gnu.org/software/wget/wget.html, [Accessed Nov 8th, 2013].

[14] Amazon, "AWS SDK for Java," http://aws.amazon.com/sdkforjava/, [Accessed Nov 11, 2013].

[15] A. K. Dutta and R. Hasan, "How much does storage really cost? towards a full cost accounting model for data storage," in *Economics of Grids, Clouds, Systems, and Services*. Springer, 2013, pp. 29–43.

[16] Amazon, "Amazon ec2 pricing," http://aws.amazon.com/ec2/pricing/, [Accessed Nov 12th, 2013].

[17] ——, "Amazon SQS pricing," http://aws.amazon.com/sqs/pricing/, [Accessed Nov 11, 2013].

[18] Y. Pan and X. Ding, "Anomaly based web phishing page detection," in *Proceedings of 22nd Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2006, pp. 381–392.

[19] Y. Zhang, J. I. Hong, and L. F. Cranor, "Cantina: a content-based approach to detecting phishing web sites," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 639–648.

[20] M. Dunlop, S. Groat, and D. Shelly, "Goldphish: Using images for content-based phishing analysis," in *Proceedings of the 5th International Conference on Internet Monitoring and Protection (ICIMP)*. IEEE, 2010, pp. 123–128.

[21] T. Li, F. Han, S. Ding, and Z. Chen, "Larx: large-scale anti-phishing by retrospective data-exploring based on a cloud computing platform," in *Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2011, pp. 1–5.