TOWARDS A CLOUD-BASED APPROACH FOR SPAM URL DEDUPLICATION FOR BIG DATASETS

Shams Zawoad, Ragib Hasan, Gary Warner, Md Munirul Haque
University of Alabama at Birmingham
{zawoad, ragib, gar, mhaque}@cis.uab.edu

Abstract

Spam emails are often used to advertise phishing websites and lure users to visit such sites. URL blacklisting is a widely used technique for blocking malicious phishing websites. To prepare an effective blacklist, it is necessary to analyze possible threats and include the identified malicious sites in the blacklist. However, the number of URLs acquired 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. To solve the problem of massive computing and storage resource requirements, we need a highly distributed and scalable architecture, where we can provision additional resources to fetch and analyze on the fly. Moreover, there is a high degree of redundancy in the URLs extracted from spam emails, where more than one spam emails contain the same URL. Hence, preserving the contents of all the websites causes significant storage waste. Additionally, fetching content from a fixed IP address introduces the possibility of being reversed blacklisted by malicious websites. In this paper, 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 deduplicating 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 URL and accumulates the content of these unique websites in a central repository. This database and website repository will be a great resource to identify phishing websites and other counterfeit websites. We show the effectiveness of our architecture using real-life, large-scale spam-based URL data.

Keywords: URL Deduplication, Cloud, Parallel Architecture, Phishing, Spam URL.

1. INTRODUCTION

Phishing is one of the most widespread threats to Internet users, who often fall victim to 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 phishing spam emails is to draw users to visit those malicious 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 (Jakobsson, 2006).

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 ("The year in phishing", 2013). 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 ("Global phishing survey", 2012), 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 ("UAB Spam Data Mine", 2014). *bounce.io*[1] collects more than a million URLs each hour from its spam email sources. Identify malicious URLs, such as phishing website by analyzing this large number of potentially malicious URLs using the local computing resources is almost an impossible task, considering the time

[1] https://bounce.io/company

and storage it requires, and the corresponding expenses. Moreover, a good percentage of these URLs point to the same website, which introduces massive computation as well as storage wastage if we fetch the content of the duplicate websites. A cloud-based URL duduplication service can reduce the number of suspicious URLs by a great extent and thus, can make it possible to analyze this large number of URLs coming from spam emails, 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 or spammers.

To accomplish the goal, we propose CURLA, a **C**loud-based spam **URL** **A**nalyzing platform. The spam URL source that we used in this paper is prepared by *bounce.io*, an organization works in anti-spam. *bounce.io* preserves data in Amazon S3. The source controller module of CURLA fetches data from the Amazon S3 storage of *bounce.io*. Whenever a new URL appears from this data source, an Amazon Elastic Computing Cloud (EC2) ("Amazon EC2", n.d.) instance will be assigned to identify whether the URL is unique or a duplicate copy. First, a heuristic-based greedy deduplication approach is executed. For content-based matching, the instance generates MD5 hashes of the fetched files and compares with the previously stored hashes. We propose a Bloom filter-based hash-matching scheme, which improves the performance in terms of time and space significantly. For a new unique URL, the EC2 instance forwards the content of the website to the local UAB storage. Otherwise, the URL is marked as a duplicate copy of a previously found URL. CURLA provides two levels of parallelism: one is machine level parallelism, and another is thread level. While thread level parallelism is easy to achieve using local infrastructure, machine level parallelism using local infrastructure incurs massive cost. 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 reversed blacklisted (Ferguson, 2012).

Contributions. The contributions of this paper are as follows:

- We propose CURLA, a cloud-based distributed and parallel architecture to deduplicate large number of potentially malicious URLs.
- CURLA removes the duplicate websites to rapidly identify malicious websites (e.g., phishing) and reduces the local storage cost since we no longer need to store the content of the duplicate websites.
- As service provided by cloud computing platforms can be scaled up easily, CURLA can be effectively scaled up to deduplicate large number of URLs by leveraging the scalability provided by cloud computing platforms. Based on the workload, we can easily add more computing nodes with little configuration.
- We propose a Bloom filter-based website matching algorithm, which provides better space and time complexity compared to traditional matching procedures.
- We develop CURLA-AWS Manager, an easy to use application to bootstrap and manage the whole URL deduplication task. A URL analyst can use this application to easily deploy and manage a highly scalable cloud-based distributed and parallel URL deduplication infrastructure.
- We evaluate the feasibility of CURLA by deploying a cloud infrastructure consisting of 100 Amazon EC2 instances and executing the deduplication service on the live data provided by *bounce.io*.
- We propose some performance optimization techniques, such as greedy analysis and use of RAM disks and show the effectiveness of these schemes. These schemes can also be used in other URL analysis tasks.

Organization. The rest of the paper is organized as follows: Section 2 discusses the motivation behind this work. In Section 3, we provide the architecture, and workflow of the system. Section 4 provides the implementation and experimental result. Section 5 discusses the effects of utilizing CURLA on different aspects of counterfeit website detection. Section 6 presents a review of related work and finally, we conclude in Section 7.

2. RESEARCH MOTIVATION

Phishing websites are replicas of legitimate web sites such as on-line banking, auction, or e-mail pages. These counterfeit websites are then deployed on publicly accessible locations, by either acquiring web hosting space or exploiting vulnerable and compromised web servers. Users are lured by spam emails to visit the phishing websites and provide confidential information, such as usernames, passwords, and pin. This information is stored for later use or resale to third parties [1]. Hence, a successful phishing attack consists of two parts: first, the creation of a fake website, usually imitating the login page of a financial institution or other online service; 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 a blacklist of phishing websites, we need to know which websites are real phishing websites and which are legitimate websites. URLs collected from spam emails are 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, if not impossible, problem because of the amount of time, labor, and expense it requires.

To overcome the problem of manual review, researchers proposed several content-based phishing website detection methods for automating the determination of phishing websites (Wardman, 2008), (Wardman, 2011), (Whittaker, 2010). 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 a website's content requires significant amount of time, the incident response time will be high if we cannot fetch and analyze large number of websites simultaneously. Second, a significant portion of URLs point to the same website. Hence, for duplicated URLs, both computation resources and storage resources are wasted. 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 can 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 with fixed public IP addresses, which can be blocked by malicious websites after a few attempts. Using a cloud-based distributed architecture, we can achieve desired performance at a low cost, and can also change the public IP address of the cloud instances dynamically for not being reversed blacklisted.

3. CLOUD-BASED SPAM URL ANALYZER

The two main objectives of CURLA are to improve the performance of URL deduplication by introducing parallelism, and to reduce the local storage cost by storing only unique websites' content. 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.

3.1 ARCHITECTURE

The system is built using the following modules:

- Controller
- Fetcher Message Queue
- Fetcher
- Uploader Message Queue
- Uploader
- URL Database
- Central website repository

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

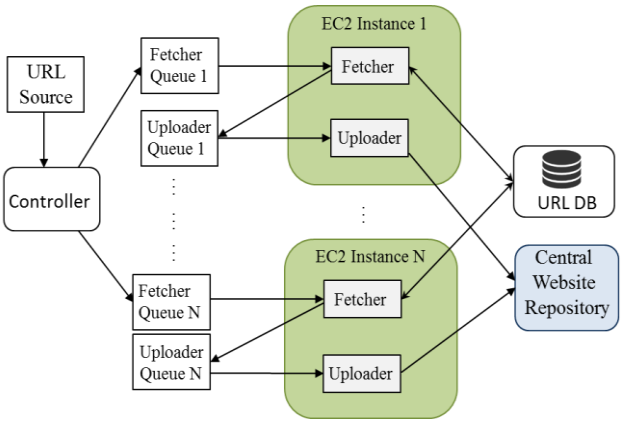

Figure 1. CURLA System Architecture

Controller. The controller module triggers the total operation of the system. This module is responsible to fetch and parse the data of bounce.io from their Amazon S3

Storage and send URL-fetch-request message to a fetcher queue selected by round-robin fashion. This module is composed of three different components: an Amazon S3 object fetcher, an URL parser, and the SQS message sender. Control flow of these components is illustrated in Figure 2.

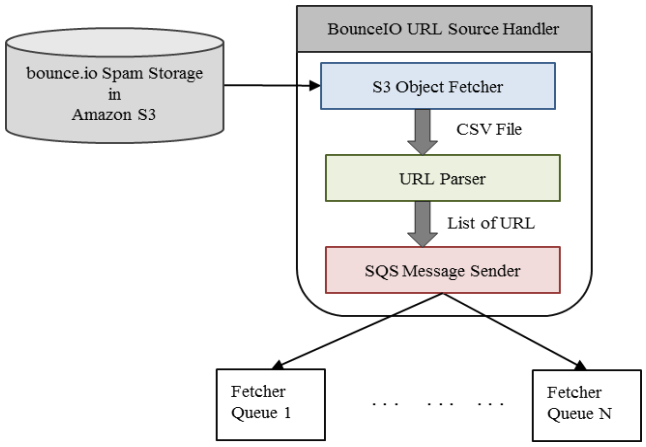

Figure 2. Process Flow of Controller Module

bounce.io collects spam emails and store the spam email information including the URLs exist in the emails in the Amazon S3 storage. They store the data in csv and json format and publish the data in hourly basis. The S3 object fetcher downloads the latest hourly file that contains information of spam emails of that hour and is in csv.gz format. Then it decompresses the downloaded file and sends the csv file to the URL parser component. The URL parser component extracts URLs from the csv file and sends a list of URLs to the SQS message sender component. To avoid overloading the memory, the URL parser component accumulates a fixed limited number of URLs and sends to the SQS message sender. After receiving a list of URLs from the URL parser, the SQS message sender sends URLs (messages) to fetcher queues in the round-robin style.

Fetcher Message Queue. Each of the fetcher message queues is an instance of Amazon Simple Queuing Service (SQS) ("Amazon SQS", n.d.). 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 Amazon SQS API provides the queuing functionality.

Fetcher. Fetcher module runs on an Amazon EC2 instance, which we refer as analyzer 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. If the website is found as new, then it sends a website-upload message to uploader message queue. This module is capable of handling multiple URL-fetch-requests simultaneously. The task of fetching and analyzing one website can be encapsulated into one thread and after receiving a message from the fetcher message queue, the fetcher can spawn 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. One 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 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 (Bloom, 1970) based algorithm to match the uniqueness of two websites. Hence, for each unique URL, we store only one Bloom filter 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, which accepts FTP connection from the uploader modules.

3.2 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 fetching and analyzing task for the received URL. If the website is found as new, a website-upload request message is

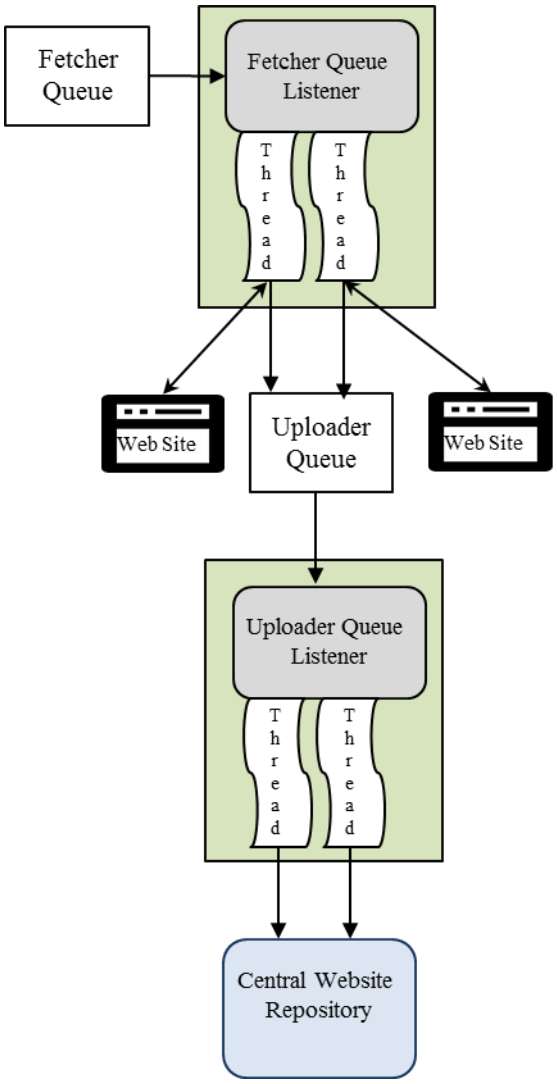

Figure 3. Work Flow of CURLA

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 workflow of CURLA is depicted in Figure 3.

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.

Algorithm 1: Website Matching Using Bloom Filter

```
1: bloomFilter  Bloom filter with 0.01% false possitive
2: probability for 500 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
```

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* ("Gnu project", n.d.) with one level of recursive fetching, time-out, ignore robots, and user agent option.

A naive approach of analyzing a website's content for finding duplicate website is to generate the hash of all the files of the website and matches 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*, 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)*.

We developed a Bloom filter based matching algorithm, which reduce 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

(Bloom, 1970). 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, we can reduce the storage cost from *O(np)* to *O(n)*.

At the time of matching the content of a newly fetched website, the analyzer selects one 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 websites exist 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.

After analyzing the content of a website, if a 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 an 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.

4 IMPLEMENTATION AND EVALUATION

In this section, we provide implementation, an overview of the working data set, experimental setup, evaluation, and some performance optimization schemes for CURLA.

4.1 IMPLEMENTATION

We developed CURLA AWS Manager, a Java-based desktop application, which manages the whole operation of the proposed system by controlling the required Amazon EC2 instances and SQS queues. We present the major modules of the application below.

Fetcher and Uploader Queue Manager. This module is responsible to manage the Amazon SQS Queues. The fetcher queue manager panel is presented in Figure [4]. Features provided by this module are:

- *Create Queues:* Using this feature, we can create any number of fetcher or uploader queues depending on number of instances we want to use. This feature is used at the system initialization and later, when we need to add one or more new instance(s) on the fly depending on the load of the system.
- *Clear Queues:* This feature is used to clear all the messages stored in a selected queue or to clear messages of all the fetcher or uploader queues. This feature is required to re-initiate one or more instances.
- *Remove Queues:* To trouble-shoot and to re-launch a new instance or the whole system, we may need to remove existing queue(s). This feature supports this requirement.

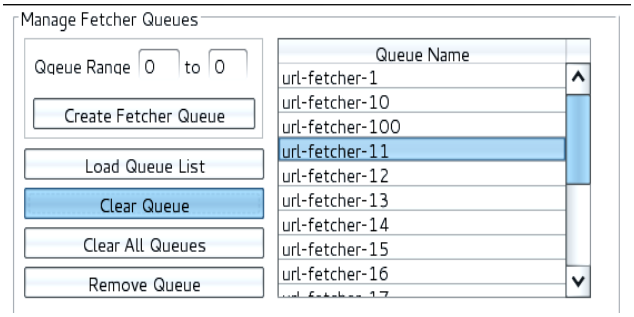

Figure 4. Fetcher Queue Manager Panel

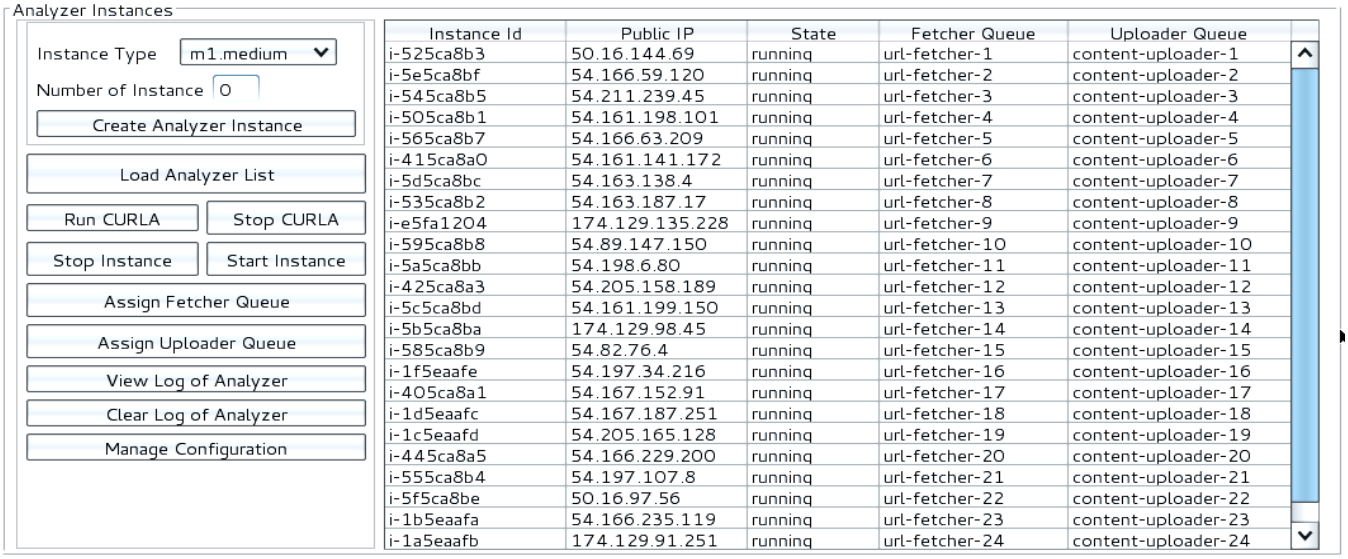

Figure 5. Analyzer Instance Manager Panel

EC2 Instance Manager. This module is used to manage the EC2 instances and run CURLA on the instances. The control panel of this module is illustrated in Figure 5. We present the notable features of this module below.

- *Create instances:* We created an Amazon Machine Images (AMI) with all the required system configuration and the analyzer module installed. This feature provides support for creating any required number of instances using the pre-build AMI and chosen instance type (e.g., m1.small, m3.large, etc.).

- *Start/Stop Instance:* Using these features, we can start and stop selected instances from a list of already created instances.
- *Assign Fetcher and Uploader Queue:* Using this functionality, we can attach a previously created fetcher and uploader queue to an analyzer instance.
- *Start/Stop CURLA*: These features provide us the functionality of executing the URL deduplication task of CURLA in the remote EC2 instances. After executing, the fetcher and uplaoder components listen to the fetcher and uploader queues that are selected in the previous step. We can also stop CURLA running on an EC2 instance.
- *Log Viewer:* This features provides an interface to the end users to view the logs of the program running in EC2 instances

S3 Source Controller. For Amazon S3-based URL source, we prepared another AMI, which downloads contents from S3 and push the URLs to the fetcher queues. Using this component of the CURLA AWS Manager, we can start/stop the operation of the source controller and can change various configurations of the source controller program.

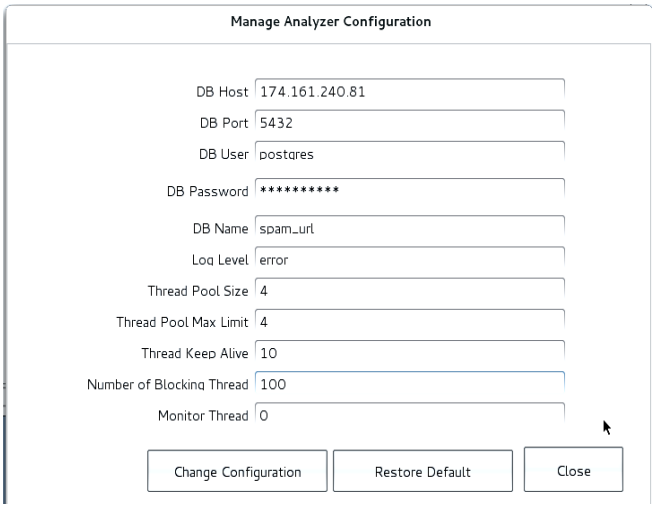

Figure 6. Configuration Manager for Analyzer

Configuration Manager. This module is used to configure different parameters of the analyzer program running inside an EC2 instance. For example, information about the database server that preserves the unique websites, log level, maximum number of threads, etc. can be configured using this module. All the EC2 instances will use these configurations while executing their operations. The configuration manager panel is illustrated in Figure 6.

4.2 EVALUATION

Data Set. The data set for this research is collected from bounce.io. Table 1 represents a summary of hourly data of seven consecutive days (October 23rd to October 29th 2014). From Table 1, we notice that on an average we found more than 50 million URLs on each day. A good percentage of these unique URLs are redundant, which we can only identify after analyzing the URLs.

Table 1. Data Set

Day	Hourly Information (No. Of URLs in Million)			Total No. Of URLs (Million)
	Average	*Min*	*Max*	
10/23/2014	4.08	2.56	7.34	97.98
10/24/2014	3.78	2.67	5.22	88.74
10/25/2014	2.29	1.67	3.26	55.08
10/26/2014	2.17	1.27	5.39	52.06
10/27/2014	3.85	2.36	5.19	92.44
10/28/2014	4.79	2.53	10.37	115.04
10/29/2014	3.89	2.67	5.75	93.42

System Configuration. In our experiments, we used an Amazon m1.medium instance as the S3 source controller. The controller program is built on OpenJDK (version:1.6.0_27). We used AWS SDK for Java ("AWS SDK", n.d.) to manage the fetcher and uploader message queues. We used 100 Amazon EC2 medium instances (m1.medium) to fetch, analyze and upload the unique content to central repository. Each of the instances was running Ubuntu 12.04.2 LTS operating system. As reported by the OS, the CPU of a m1.medum instance is single core Intel(R) Xeon(R) CPU E5645 (12MB cache, 2.40GHz) with 3.7GB 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 (c3.large) as a central repository of the unique websites and URL database. The OS of the instance is Ubuntu 12.04.2 LTS, and the CPU has dual Intel(R) Xeon(R) E5507 (4MB cache, 2.27GHz) processors. To upload the website content, we used FTP. Hence a FTP server was installed in this machine. The URL database was maintained by PostgreSQL 9.1.9.

Deduplication Results. In our experiment, we run 100 Amazon EC2 instances for 12 hours to analyze two hours of URLs collected from the S3 storage of bounce.io. We used 4 threads on each instances. The most recent URL dataset available at the time of experiment was used. In our first attempt, we run the experiment for the URLs reported at 6:00 am (Time Zone +00) on October 28th. In the second attempt, we run the experiment for the URLs available at 8:00 am on October 30th. Table 2 presents some statistics of the URL deduplication results. As we notice from Table 2, we failed to fetch the content for a large number of URLs. A possible reason is that the websites went down before fetching. Since we ignore the robots.txt file in wget, the wget program is not blocked because of the robots.txt. However, after introducing the user agent option with wget, the percentage of URLs that were failed to fetch got reduced from 90% to 69% in the second attempt. As we notice from the Table 2, this higher success rate in fetching helped us to find higher number of duplicate URLs. Table 2 also represents the amount of storage that we can save by deduplicating URLs. According to the storage cost model of (Dutta, 2013), cost of the storage saved in trial one is $3.86 USD and for trial 2 is $18.35 USD. Obviously there will be a significant amount of cost savings after deduplicating all the URLs of a day.

Table 2. Deduplication Results

Trial	No. Of Unique URLs	No. Of Duplicate URLs	No. Of URLs failed to Fetch	Storage Saved (GB)
1	22445	12552	353403	4.85
2	24318	42846	148942	23.05

Fetch Time. To calculate the fetch time of a website, we measured the fetch time of 5000 unique URLs in five different trials; where in each trial, we used 1000 unique URLs. These unique websites were found from our first attempt of URL deduplication. Fetch time of each trial is presented in Figure 7. The averages of each trial were ranged between 6 to 8 seconds. From the average of each trial, we measured the cumulative average fetch time of all the 5000 URLs as 6.8 seconds. Similarly the cumulative standard deviation for the fetch time is 11 seconds.

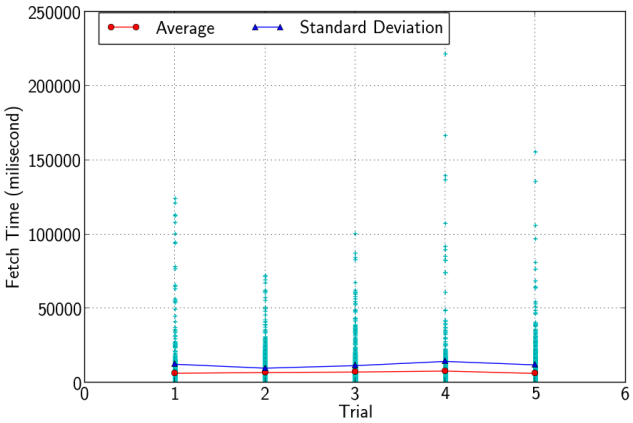

Figure 7. Determining URL Fetch Time

Analyze Time. To calculate the analyze time of a website, we measured the average analyze time of every 1000 unique URLs found in the first attempt of URL deduplication. The results of this experiment are illustrated in the Figure 8.

From the Figure 8, we notice that the time required for URL analysis increases linearly with the increase in number of URLs. According to linear regression analysis, the best-fit function that matches with the experimental results is as follows:

Analyze Time = 5.92 Number of URL – 14744.10* (1)

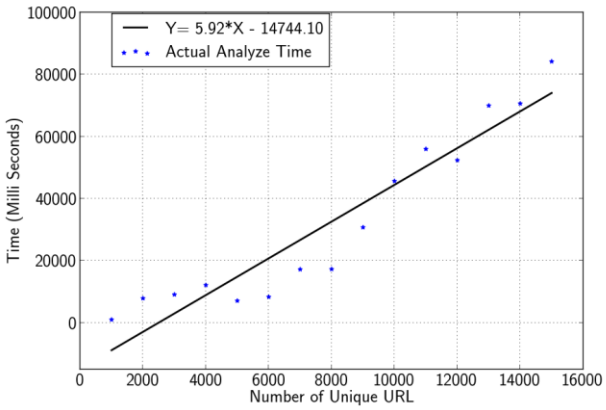

Figure 8. Determining URL Analyze Time

Upload Request Time. To calculate the upload request time of a website, we used the same approach that we did for fetch time calculation. We measured the upload request time of 5000 unique URLs in five different trials. Upload request time of each trial, where in each trial 1000 unique URLs are chosen is presented in Figure 9. From the average of each trial, we measured the cumulative average upload request time of all the 5000 URLs as 0.5 seconds. Similarly the cumulative standard deviation for the upload request time is 0.1 seconds.

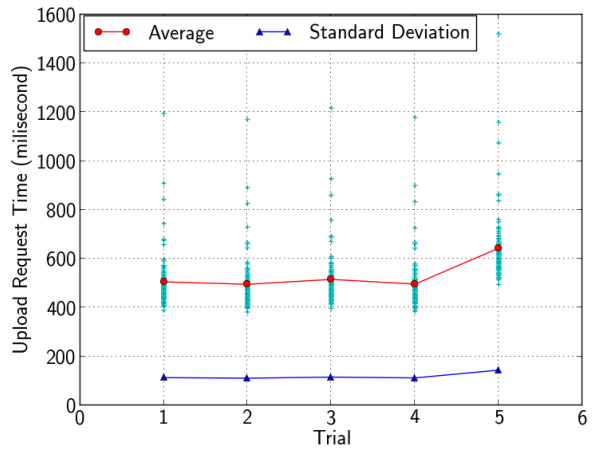

Figure 9. Determining URL Upload Request Time

Time Estimation. Using the calculated URL fetch, analyze, and upload request time, we can estimate the required time to analyze a given set of URLs for a certain number of EC2 instances and number of threads running in each instance. We used the average fetch time and upload request time while estimating the total time. For analyze time, the time increases linearly with the number of URLs. In this case, we considered the multiplication coefficient of the regression equation 1. We derive the regression equation for analyze time using the average time required for every 1000 URLs. Hence, we assume that for every one thousand URLs, the analyze time increases with factor of 5.92.

Suppose, there are N thousand of URLs that are successfully analyzed. F is the number of URLs that we fail to fetch and *WT* is the timeout value used with *wget*. Now, if we run M number of m1.medium EC2 instance with H number of thread, then the average required time T can be calculated as follows:

$$T = \frac{N*1000*(TF+TR)}{M*H} + \frac{5.92*N*(N+1)*1000}{2*M*H} + \frac{F*WT}{M*H} \quad (2)$$

Here, *TF* and *TR* are the average fetch and upload request time respectively and as calculated previously, *TF*=6.8 and *TR* = 0.5 seconds.

We apply the Equation 2 on the deduplication results found in the second attempt and measure the accuracy of the Equation 2. In the second attempt, N = 67.164, F = 148942, WT = 20 seconds, M = 100, and H = 4. Hence, according to the Equation 2, total required time T = 42551.28 seconds. In the second attempt, we actually run the analyzing task for 12 hours, which is 43200 seconds. Hence, using equation 2, we can estimate the required time with 98.5% accuracy. Considering 11 seconds average standard deviation in the fetch time, this accuracy is satisfactory.

Bloom Filter-based Analysis. Performance improvement of Bloom filter-based matching algorithm is depicted in Figure 10, 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.

4.3 *Optimization*

Greedy Analysis. While deduplicating URLs by analyzing contents, we found that among 12,552 duplicate URLs of trial 1, 91.6% of those duplicate URLs have the

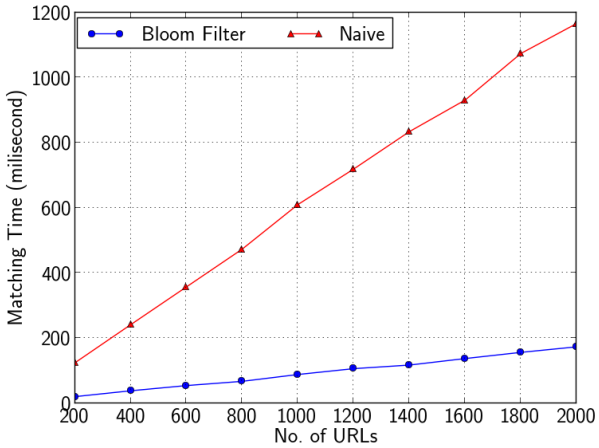

Figure 10. Performance Analysis of Bloom Filter-based Matching Algorithm

same top domain as its counterpart unique URLs. This finding motivates us to use a greedy analysis approach by avoiding fetching a URL if the top domain of the URL is previously seen in the list of already found unique URLs. By top domain, we refer the TLD (top-level domain) including the last subdomain. For example, if a URL is http://appl.businessstandard.in, the top domain for this URL is businessstandard.in.

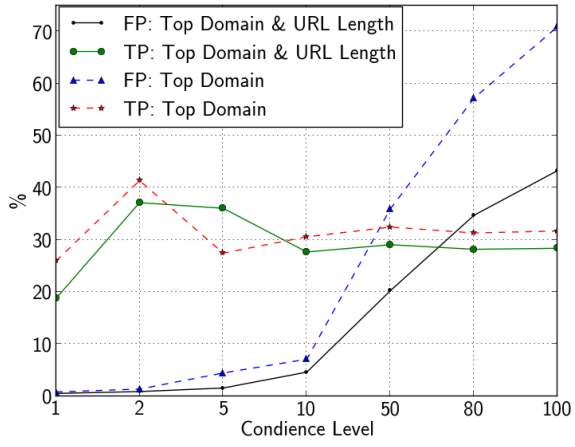

Figure 11. Performances of Two Different Heuristics

However, there are also significant portions of unique URLs, which have the same top domains. Among 22445 unique URLs, on average on top domain is found in 8 unique websites. Hence, we used a probabilistic method to decide whether we should fetch a URL if the top domain is already seen or we determine the URL as a duplicate copy. If a top domain is previously seen, we will determine the URL as duplicate with some certain probability. This probability is referred as confidence level. To test this hypothesis, we used a set of URLs consisting of 5000 URLs, which are already deduplicated by analyzing their content. We apply our hypothesis on this dataset for different confidence level. Figure 11 represents the % of true positive (TP) and % of false positive (FP) rate for determining duplicate URLs using the greedy analysis. From the Figure 11, we notice that %FP increases significantly with the increase in confidence level. This increase in %FP means we will lose higher number of unique URLs, which will be marked as duplicate.

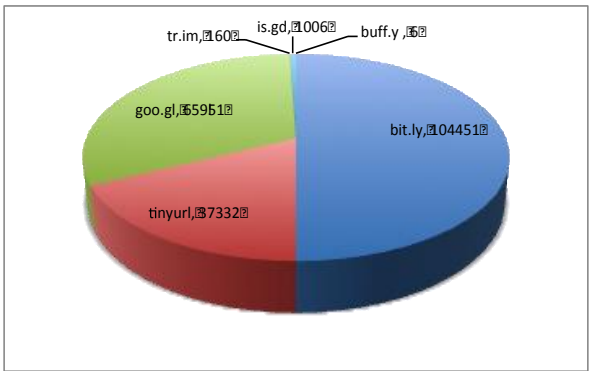

Figure 12. Distributions of URL Shortener Services on October 28th, 2014

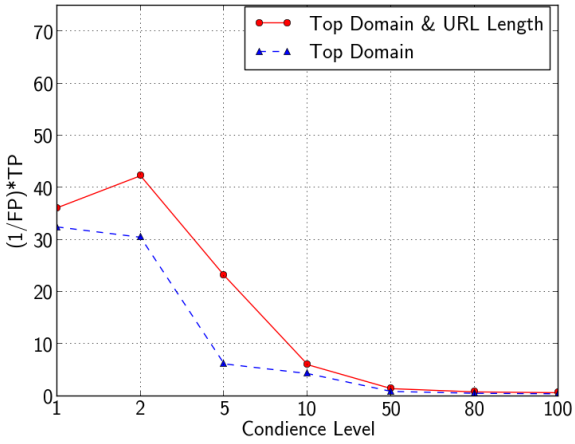

Figure 13. Selections of the Optimal Heuristic

To improve the performance of the greedy analysis, we first consider the top domains that come from the URL shortener services. Figure 12 represents the distribution of various URL shortening domains of October 28th. This is quite obvious that for URL shorteners, same top domain can point

to different websites. Hence, applying probabilistic behavior on the top domains, which come from URL shorteners will increase the %FP. Moreover, besides similar top domain, equal length of a URL can give us more confidence that the URL can be a duplicate website. After including these two attributes to our hypothesis we were able to reduce the %FP significantly, which is presented in Figure 11.

While using such greedy analysis, the goal should be reducing %FP and increasing %TP. To identify the best configuration for the greedy analysis, we apply the same criteria. In Figure 13, we plot (1/FP)*TP for the two configurations using different confidence level. From Figure 13, we identify that 2% confidence level using top domain and URL length is the best heuristic for the greedy analysis.

IO Optimization Using RAM Disk. RAM disk is a part of the RAM, which the operation system treats as a secondary storage. The content of a website that we download to analyze will be eventually removed after the uploader module uploads the content to the central repository. Hence, we do not need a persistent storage to store the content of a website. Moreover, since the performance of RAM is significantly better than the secondary storage, we used RAM disk to store the content of a website.

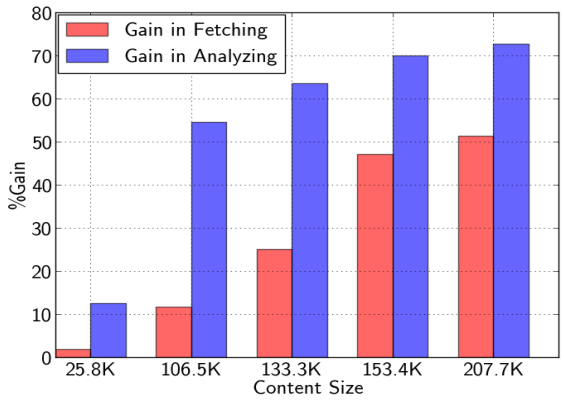

Figure 14. Performance analysis of RAM Disk

To identity the performance of RAM disk in URL analysis, we select five different URLs with varying content size. We execute the fetching and analysis task 100 times for each URL using RAM disk and without RAM disk. We found that the performance of RAM disk is better for both URL fetching and analysis. The performance gain also increases with the increase in content size. Results of this experiment are reported in Figure 14.

5 DISCUSSION

In this section, we discuss effects on the performance of detecting phishing websites, and usage of CURLA in areas other than phishing.

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. 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 24 hours of computing time for 86400 duplicate websites.

Another impact on phishing detection is that it will help to increase the rate of phishing identification. 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 in the spam email, identifying pattern of email subject, etc. However, the phishers are also improving in writing the content of spam emails, or frequently change the structure of the spam emails to spoof this technique. There can be a spam email containing a phishing URL, which does not have any content related to the phishing website. 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 our system, we can improve the rate of phishing website identification. Therefore, by providing only the new unique websites to phishing analyst, we can significantly improve the productivity of the analyst

Other Application Area. In addition to detecting phishing websites, CURLA can be effectively used in identifying other types of counterfeit websites, which are also advertised by spam emails. For example, the counterfeit websites for goods constitute a significant portion of websites that we fetched. 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 that sell their products, so that they can 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. Using our approach, we can provide unique content to the analyst, who can expedite the process of identifying these counterfeit websites.

6 RELATED WORK

Content based phishing website detection has been explored by researchers and there are various established approaches (Wardman, 2008), (Wardman, 2011), (Whittaker, 2010), (Zhang, 2007), (Dunlop, 2010). 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 To expedite the process of phishing website detection, Wardman et al. proposed content-based matching algorithm, Deep MD5. The basis of this approach is that, by considering the MD5 values of files that make up the human experience of the webpage, e.g., .jpg and .png, .gif, .css, and .js files, we may find large enough similarities to identify a phishing website even when an MD5 matching between HTML files may be failed due to minor textual changes on the HTML files. Pan et al. used an identity extractor to determine the legitimacy of a website using textual clues, such as words that appear in the DOM's title, address, and body (Pan, 2006). Xiang and Hong proposed a similar identity extraction algorithm that employs information extraction and retrieval algorithms to differentiate websites that is claiming to be a particular website versus legitimate websites (Xiang, 2009). CANTINA, a content-based solution (Zhang, 2007), can accurately identify roughly 95% of phishing sites with a false positive rate of only 6%. This approach must use the content from the webpage, not only the URL. However, none of these solutions use cloud-based content fetching approach.

Martignoni et al. proposed a cloud-based framework for dynamic behavior based malware analysis (Martignoni, 2009). Our CURLA framework utilizes the computational power available in clouds with the heterogeneity of end-users' environments. CURLA allows an end-user to forward the execution and analysis of a potentially malicious program to cloud environment and force the program to behave as if it were executed directly in the environment of the end-user. Hence, it provides the facility of monitoring the execution of a potentially malicious program in a realistic end-user's environment and also raises the end-user's protection level by leveraging the computational resources of clouds for fine-grained analysis that would not be feasible otherwise.

Li et al. proposed a cloud-based offline phishing detection system named LARX, which uses network traffic data archived at a vantage point and analyzes these data for phishing attacks (Li, 2011). 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. Due to scalable nature of clouds, LARX can be easily scaled up to analyze a large volume of trace data when enough computing power and storage capacities are provided. They used two existing cloud platforms, Amazon Web Services and Eucalyptus.

The closest work related to our work is conducted by Ferguson et al. in (Ferguson, 2012), 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 (Ferguson, 2012). 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 (Ferguson, 2012).

7 CONCLUSION AND FUTURE WORK

In this paper, we have presented CURLA, a highly scalable and distributed cloud-based framework for deduplicating big datasets of URLs collected from spam emails. Creating and maintaining an equivalent local infrastructure 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 results show the effectiveness of CURLA on a very large 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 helps to improve the rate of phishing and other types of counterfeit websites detection. The greedy analysis approach and use of RAM disk might be useful in other type of URL analysis tasks besides URL deduplication. Finally, the CURLA-AWS Manager along with the analyzer AMI can be used by a URL analyst to deploy a cloud-based highly scalable URL deduplication infrastructure very conveniently.

One of the problems that we faced in our current work is that the *wget* program failed to fetch a good percentage of websites' content. It can happen if a malicious website went down before we start fetching. However, in normal situation, we can expect a website to remain live for several hours. Hence, we need to investigate this issue more to increase the rate of successful fetching. 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 a large-scale spam URL datasets.

8 ACKNOWLEDGMENT

This research was supported by a Google Faculty Research Award, the Office of Naval Research Grant #N000141210217, and the Department of Homeland Security Grant #FA8750-12-2-0254. Thanks to bounce.io for providing us the access of their spam email data and Jason Britt to help us parsing the bounce.io data.

9 REFERENCES

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

The year in phishing (2013). Retrieved December 11, 2014,from http://www.emc.com/collateral/fraud-report/online-rsa-fraud-report-012013.pdf.

Global phishing survey: Trends and domain name use in 1H2012 (2012). Retrieved December 11, 2014, from http://anti-phishing.org/reports/APWGGlobalPhishingSurvey 1H2012.pdf.

UAB Spam Data Mine (2014). Retrieved December 11, 2014, from https://cis.uab.edu/uab-spam-data-mine/.

Amazon elastic compute cloud (amazon ec2).(n.d.). Retrieved December 11, 2014, from http://aws.amazon.com/ec2/.

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

Wardman, B., Warner, G. (2008). Automating phishing website identification through deep MD5 matching, *Proceedings of IEEE eCrime Researchers Summit*, 1–7.

Wardman, B., Stallings, T., Warner, G., Skjellum, A. (2011). High performance content-based phishing attack detection, *Proceedings of IEEE eCrime Researchers Summit*,1–9.

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

Amazon simple queue service (Amazon SQS). (n.d.). Retrieved December 11, 2014, from http://aws.amazon.com/sqs/.

Bloom, B. (1970). Space/time trade-offs in hash coding with allowable errors, *Communications of the ACM*, 13(7), 422–426.

Gnu project free software foundation(FSF). (n.d.). Retrieved December 11, 2014, from http://www.gnu.org/software/wget/wget.html.

AWS SDK for Java. (n.d.). Retrieved December 11, 2014, from http://aws.amazon.com/sdkforjava/.

Dutta, A. K., Hasan, R.(2013). How much does storage really cost? Towards a full cost accounting model for data storage, *Proceedings of Economics of Grids, Clouds, Systems, and Services*, 29–43.

Zhang, Y., Hong, J. I., Cranor, L. F. (2007). Cantina: a content-based approach to detecting phishing web sites, *Proceedings of 16th ACM international conference on World Wide Web*, 639–648.

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

Pan, Y., Ding, X. (2006). Anomaly based web phishing page detection, *Proceedings of 22nd IEEE Annual Computer Security Applications Conference*, 381–392.

Xiang, G., Hong, J. I. (2009). A hybrid phish detection approach by identity discovery and keywords retrieval, *Proceedings of the 18th ACM international conference on World wide web*, 571–580.

Martignoni, L., Paleari, R., & Bruschi, D. (2009). A framework for behavior-based malware analysis in the cloud. *Proceedings of Information Systems Security*, 178-192.

Li, T., Han, F., Ding, S., & Chen, Z. (2011). LARX: Large-scale anti-phishing by retrospective data-exploring based on a cloud computing platform. *Proceedings of the 20th IEEE International Conference on Computer Communications and Networks*, 1-5.

Authors

Shams Zawoad is a PhD student and a graduate research assistant at the Computer and Information Sciences Department of University of Alabama at Birmingham (UAB). His research interest is in cloud forensics, anti-phishing, and secure provenance. He received his B.Sc. in Computer Science and Engineering from Bangladesh University of Engineering and Technology (BUET) in January 2008.

Dr. Ragib Hasan is a tenure-track Assistant Professor at the Department of Computer and Information Sciences at the University of Alabama at Birmingham. Prior to joining UAB, He received his Ph.D. and M.S. in Computer Science from the University of Illinois at Urbana Champaign in October, 2009, and December, 2005, respectively, and was an NSF/CRA Computing Innovation Fellow post-doc at the Department of Computer Science, Johns Hopkins University. Hasan has received multiple awards in his career, including the 2014 NSF CAREER Award, 2013 Google RISE Award, and 2009 NSF Computing Innovation Fellowship.

Gary Warner is the director of research in Computer Forensics at the University of Alabama at Birmingham (UAB) and also the Chief Technologist and Co-Founder of Malcovery Security. He was the founding president of the Birmingham chapter of the FBI's InfraGard program. He has been recognized by FBI Director Robert Mueller for "Exceptional Service in the Public Interest" and received the IC3 and NCFTA's Partnership Award "in recognition of his outstanding support in the ongoing battle against cybercrime".

Md Munirul Haque, Ph.D., is a post-doctoral fellow in SECRETLab at the University of Alabama at Birmingham. Previously, he was a member of Ubicomp lab, Marquette University, USA, from where he received his Ph.D. in 2013. He has received his B.Sc degree in Computer Science and Engineering from Bangladesh University of Engineering and Technology (BUET), Bangladesh in 2003.