Design and Evaluation of a Real-Time URL Spam Filtering Service

Kurt Thomas*, Chris Grier*†, Justin Ma*, Vern Paxson*†, Dawn Song*
{*kthomas, grier, jtma, vern, dawnsong*}*@cs.berkeley.edu*
* University of California, Berkeley † International Computer Science Institute

Abstract

On the heels of the widespread adoption of web services such as social networks and URL shorteners, scams, phishing, and malware have become regular threats. Despite extensive research, email-based spam filtering techniques generally fall short for protecting other web services. To better address this need, we present Monarch, a real-time system that crawls URLs as they are submitted to web services and determines whether the URLs direct to spam. We evaluate the viability of Monarch and the fundamental challenges that arise due to the diversity of web service spam. We show that Monarch can provide accurate, real-time protection, but that the underlying characteristics of spam do not generalize across web services. In particular, we find that spam targeting email qualitatively differs in significant ways from spam campaigns targeting Twitter. We explore the distinctions between email and Twitter spam, including the abuse of public web hosting and redirector services. Finally, we demonstrate Monarch's scalability, showing our system could protect a service such as Twitter—which needs to process 15 million URLs/day—for a bit under $800/day.

1. Introduction

In recent years, the Internet has seen a massive proliferation of web services, including social networks, video sharing sites, blogs, and consumer review pages that draw in hundreds of millions of viewers. On the heels of the widespread adoption of these services, phishing, malware, and scams have become a regular threat [1]–[3]. Bypassing protection mechanisms put in place by service operators, scammers are able to distribute harmful content through the use of compromised and fraudulent accounts [4], [5]. As spam evolves beyond email and becomes a regular nuisance of web services, new defenses must be devised to safeguard what is currently a largely unprotected space.

While email spam has been extensively researched, many of the solutions fail to apply to web services. In particular, recent work has shown that domain and IP blacklists currently in use by social network operators and by URL shortening services [6]–[9] perform too slowly (high latency for listing) and inaccurately for use in web services [5], [10], [11]. Alternative solutions, such as account-based heuristics that are specifically designed to identify automated and suspicious behavior in web services [12]–[14], focus on identifying accounts generated by spammers, and thus have limited utility in detecting misuse of compromised accounts. They also can incur delays between a fraudulent account's creation and its subsequent detection due to the need to build a history of (mis-)activity. Given these limitations, we seek to design a system that operates in *real-time* to limit the period users are exposed to spam content; provides *fine-grained* decisions that allow services to filter individual messages posted by users; but functions in a manner *generalizable* to many forms of web services.

To this end we design Monarch, a real-time system that crawls URLs as they are submitted to web services and determines whether the URLs direct to spam content. For our study, we define spam to include scams advertising pharmaceuticals, adult content, and other solicitations, phishing that attempts to capture account credentials, and pages attempting to distribute malware. By restricting our analysis to URLs, Monarch can provide spam protection regardless of the context in which a URL appears, or the account from which it originates. This gives rise to the notion of *spam URL filtering as a service*. Monarch frees other web services from the overhead of reinventing spam classifiers and the accompanying infrastructure components.

The architecture of Monarch consists of three core elements: a front-end that accepts URLs submitted by web services seeking a classification decision, a pool of browsers hosted on cloud infrastructure that visits URLs to extract salient features, and a distributed classification engine designed to scale to tens of millions of features that rapidly returns a decision for whether a URL leads to spam content. Classification builds upon a large foundation of spam characteristics [15]–[24] and includes features drawn from the lexical properties of URLs, hosting infrastructure, and page content (HTML and links). We also collect new features including HTTP header content, page frames, dynamically loaded content, page behavior such as JavaScript events, plugin usage, and a page's redirection behavior. Feature collection and URL classification occur at the time a URL is submitted to our service, with the overall architecture of Monarch scaling to millions of URLs to satisfy the throughput expected of large social networks and web mail providers.

In this paper, we evaluate the viability of Monarch as a real-time filtering service and the fundamental challenges that arise from the diversity of web service spam. We show that Monarch can provide accurate, real-time protection, but that the underlying characteristics of spam do not generalize across web services. In particular, we leverage Monarch's feature collection infrastructure to study distinctions between 11 million URLs drawn from email and Twitter. We find that spam targeting email is qualitatively different from Twitter

spam, requiring classifiers to learn two distinct sets of rules to ensure accuracy. A basic reason for this distinction is that email spam occurs in short-lived campaigns that quickly churn through spam domains, while spam on Twitter consists of long lasting campaigns that often abuse public web hosting, generic redirectors, and URL shortening services.

Our evaluation also includes an analysis of which URL features serve as the strongest indicators of spam and their persistence as spam evolves. We find that classification requires access to every URL used to construct a landing page, HTML content, and HTTP headers to ensure the best accuracy. In contrast, relying solely on DNS entries or the IP address of spam infrastructure achieves much less accuracy. Furthermore, without regular retraining and access to new labeled spam samples, accuracy quickly degrades due to the ephemeral nature of spam campaigns and their hosting infrastructure.

We deploy a full-fledged implementation of Monarch to demonstrate its scalability, accuracy, and run-time performance at classifying tweet and email spam URLs. Using a modest collection of cloud machinery, we process 638,000 URLs per day. Distributed classification achieves an accuracy of 91% (0.87% false positives) when trained on a data set of nearly 50 million distinct features drawn from 1.7 million spam URLs and 9 million non-spam URLs, taking only one hour to produce a model. While the current false positive rate is not optimal, we discuss several techniques that can either lower or ameliorate their impact in Section 6.1. During live classification, each URL takes on average 5.54 sec to process from start to finish. This delay is unavoidable and arises from network requests made the browser's crawling process itself, which is difficult to speed up; only 1% of overhead comes from instrumenting the browser for feature collection. The cloud infrastructure required to run Monarch at this capacity costs $1,587 for a single month. We estimate that scaling to 15 million URLs per day would cost $22,751 per month, and requires no changes to Monarch's architecture.

In summary, we frame our contributions as:

- We develop and evaluate a real-time, scalable system for detecting spam content in web services.
- We expose fundamental differences between email and Twitter spam, showing that spam targeting one web service does not generalize to other web services.
- We present a novel feature collection and classification architecture that employs an instrumented browser and a new distributed classifier to capture and scale to tens of millions of features.
- We present an analysis of new spam properties illuminated by our system, including abused services and redirects used to host and mask spam web content.
- We examine the salience of each feature used for detecting spam and evaluate their performance over time.

2. Architecture

In this work we present the design and implementation of Monarch, a system for filtering spam URLs in real-time as they are posted to web applications. Classification operates independently of the context where a URL appears (e.g., blog comment, tweet, or email), giving rise to the possibility of spam URL filtering as a service. We intend the system to act as a first layer of defense against spam content targeting web services, including social networks, URL shorteners, and email.

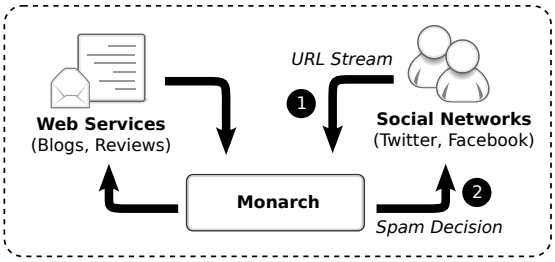

Fig. 1: Intended operation of Monarch. Web services provide URLs posted to their sites for Monarch to classify. The decision for whether each URL is spam is returned in real-time.

We show the overall intended operation of Monarch in Figure 1. Monarch runs as an independent service to which any web service can provide URLs to scan and classify. During the period it takes for Monarch's classification to complete, these services can either delay the distribution of a URL, distribute the URL and retroactively block visitors if the URL is flagged as spam (risking a small window of exposure), or employ a heavier-weight verification process to enforce even stricter requirements on false positives than are guaranteed by classification.

2.1. Design Goals

To provide URL spam filtering as a service, we adopt six design goals targeting both efficiency and accuracy:

1) *Real-time results*. Social networks and email operate as near-interactive, real-time services. Thus, significant delays in filtering decisions degrade the protected service.
2) *Readily scalable to required throughput*. We aim to provide viable classification for services such as Twitter that receive over 15 million URLs a day.
3) *Accurate decisions*. We want the capability to emphasize low false positives in order to minimize mistaking non-spam URLs as spam.
4) *Fine-grained classification*. The system should be capable of distinguishing between spam hosted on public services alongside non-spam content (i.e., classification of individual URLs rather than coarser-grained domain names).
5) *Tolerant to feature evolution*. The arms-race nature of spam leads to ongoing innovation on the part of spammers' efforts to evade detection. Thus, we require the ability to easily retrain to adapt to new features.
6) *Context-independent classification*. If possible, decisions should not hinge on features specific to a particular service, allowing use of the classifier for different types of web services.

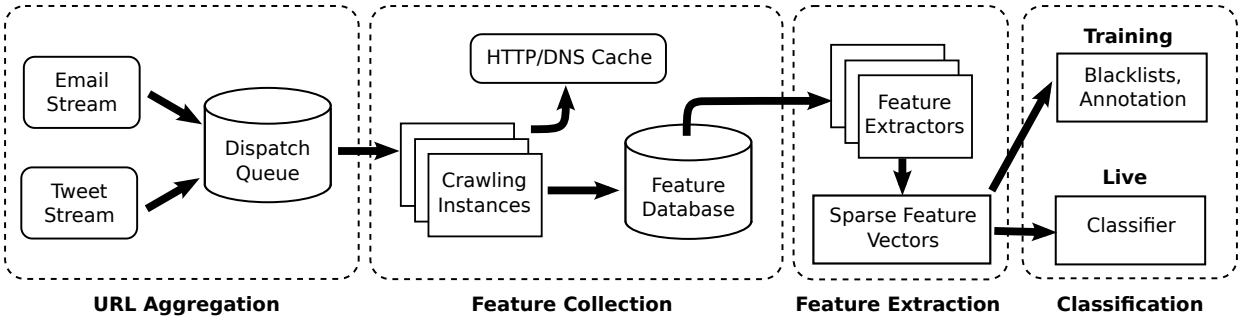

Fig. 2: System flow of Monarch. URLs appearing in web services are fed into Monarch's cloud infrastructure. The system visits each URL to collect features and stores them in a database for extraction during both training and live decision-making.

2.2. System Flow

Figure 2 shows Monarch's overall internal system flow. URLs posted to web services are fed into a dispatch queue for classification. The system visits each URL to collect its associated raw data, including page content, page behavior, and hosting infrastructure. It then transforms these raw features into meaningful boolean and real-valued features and provides these results to the classifier for both training and live decision-making. During live classification, Monarch's final decision is returned to the party that submitted the URL; they can then take appropriate action based on their application, such as displaying a warning that users can click through, or deleting the content that contained the URL entirely. We now give an overview of each component in this workflow.

URL Aggregation. Our current architecture aggregates URLs from two sources for training and testing purposes: links emailed to spam traps operated by a number of major email providers and links appearing in Twitter's streaming API. In the case of Twitter, we also have contextual information about the account and tweet associated with a URL. However, we hold to our design goal of remaining agnostic to the source of a URL and omit this information during classification. We examine how removing Twitter-specific features affects accuracy in Section 6.

Feature Collection. During feature collection, the system visits a URL with an instrumented version of the Firefox web browser to collect page content including HTML and page links, monitor page behavior such as pop-up windows and JavaScript activity, and discover hosting infrastructure. We explore the motivation behind each of these features in Section 3. To ensure responsiveness and adhere to our goal of real-time, scalable execution, we design each process used for feature collection to be self-contained and parallelizable. In our current architecture, we implement feature collection using cloud machinery, allowing us to spin up an arbitrary number of collectors to handle the system's current workload.

Feature Extraction. Before classification, we transform the raw data generated during feature collection into a sparse feature vector understood by the classification engine. Data transformations include tokenizing URLs into binary features and converting HTML content into a bag of words. We permanently store the raw data, which allows us to evaluate new transformations against it over time.

Classification. The final phase of the system flow produces a classification decision. Training of the classifier occurs off-line and independent of the main system pipeline, leaving the live decision as a simple summation of classifier weights. During training, we generate a labeled data set by taking URLs found during the feature collection phase that also appear in spam traps or blacklists. We label these samples as spam, and all other samples as non-spam. Finally, in order to handle the millions of features that result and re-train daily to keep pace with feature evolution, we develop a distributed logistic regression, as discussed in Section 4.

3. Feature Collection and Extraction

Classification hinges on having access to a robust set of features derived from URLs to discern between spam and non-spam. Previous work has shown that lexical properties of URLs, page content, and hosting properties of domains are all effective routes for classification [15], [16], [22]–[24]. We expand upon these ideas, adding our own sources of features collected by one of three components: a *web browser*, a *DNS resolver*, and *IP address analysis*. A comprehensive list of features and the component that collects them can be found in Table 1. A single monitor oversees multiple copies of each component to aggregate results and restart failed processes. In turn, the monitor and feature collection components are bundled into a *crawling instance* and replicated in the cloud.

3.1. Web Browser

Within a crawling instance, a web browser provides the primary means for collecting features for classification. Due to real-time requirements, a trade-off arises between expedited load times and fidelity to web standards. Given the adversarial nature of spam, which can exploit poor HTML parsing or the lack of JavaScript and plugins in a lightweight browser [25],

Source	Features	Collected By
Initial URL, Final URL	Domain tokens, path tokens, query parameters, is obfuscated?, number of subdomains, length of domain, length of path, length of URL (From here on out, we denote this list as URL features)	Web browser
Redirects	URL features for each redirect, number of redirects, type of redirect	Web browser
Frame URLs	URL features for each embedded IFrame	Web browser
Source URLs	URL features for every outgoing network request; includes scripts, redirects, and embedded content	Web browser
HTML Content	Tokens of main HTML, frame HTML, and script content	Web browser
Page Links	URL features for each link, number of links, ratio of internal domains to external domains	Web browser
JavaScript Events	Number of user prompts, tokens of prompts, onbeforeunload event present?	Web browser
Pop-up Windows	URL features for each window URL, number of windows, behavior that caused new window	Web browser
Plugins	URL features for each plugin URL, number of plugins, application type of plugin	Web browser
HTTP Headers	Tokens of all field names and values; time-based fields are ignored	Web browser
DNS	IP of each host, mailserver domains and IPs, nameserver domains and IPs, reverse IP to host match?	DNS resolver
Geolocation	Country code, city code (if available) for each IP encountered	IP analysis
Routing Data	ASN/BGP prefix for each IP encountered	IP analysis

TABLE 1: List of features collected by Monarch

[26], our system employs an instrumented version of Firefox with JavaScript enabled and plugin applications installed including Flash and Java. As a URL loads in the browser, we monitor a multitude of details, including redirects, domains contacted while constructing a page, HTML content, pop-up windows, HTTP headers, and JavaScript and plugin execution. We now explain the motivation behind each of these raw features and the particulars of how we collect them.

Initial URL and Landing URL. As identified by earlier research [16], [23], the lexical features surrounding a URL provide insight into whether it reflects spam. The length of a URL, the number of subdomains, and terms that appear in a URL all allow a classifier to discern between *get.cheap.greatpills.com* and *google.com*. However, given the potential for nested URLs and the frequent use of shortening services, simply analyzing a URL presented to our service does not suffice. Instead, we fetch each URL provided to the browser, allowing the browser to log both the initial URL provided as well as the URL of the final landing page that results after executing any redirects.

Redirects. Beyond the initial and final landing URL, the redirect chain that occurs in between can provide insight into whether a final page is spam. Suspiciously long redirect chains, redirects that travel through previously known spam domains, and redirects generated by JavaScript and plugins that would otherwise prevent a lightweight browser from proceeding all offer insight into whether the final landing page reflects spam. To capture each of these behaviors, the web browser monitors each redirect that occurs from an initial URL to its final landing page. This monitoring also includes identifying the root cause of each redirect; whether it was generated by a server 30X HTTP response, meta refresh tag, JavaScript event, or plugin (e.g., Flash).

Sources and Frames. In the case of mashup pages with spam content embedded within a non-spam page, the URL of a final page masks the presence of spam content. This is particularly a problem with URL shortening services, including *ht.ly* and *ow.ly*, which embed shortened URLs as IFrames. To recover information about embedded content, the web browser monitors and logs all frames, images, and ad URLs it contacts during the construction of a page. The browser also collects a list of all outgoing network requests for URLs, regardless whether the URL is for a top level window or frame, and applies a generic label called *sources*.

HTML Content. Beyond features associated with URLs, the content of a page often proves indicative of the presence of spam [24], [27], [28]. This includes the terms appearing on a page and similar layout across spam webpages. To capture page content, the web browser saves a final landing page's HTML in addition to the HTML of all subframes on the page. Naturally, we cannot collect HTML features for image-based spam or for media content such as PDFs.

Page Links. The links appearing on a final landing page offer some insight into spam. While the web browser only follows URLs that automatically load (it does not crawl embedded links such as HREFs), if a page contains a URL to a known spam page, then that can help to classify the final landing page. Similarly, search engine optimization techniques where a page comes stuffed with thousands of URLs to an external domain also suggests misbehavior. To capture both of these features, the web browser parses all links on a final landing page. Each link is subjected to the same analysis as frames and redirects. Afterwards, we compute the ratio of links pointing at internal pages versus external domains.

JavaScript Events. In addition to the content of a page, observing an attempt to force the user to interact with a page—such as pop-up boxes and prompts that launch before a user navigates away from a page—strongly indicates spam. To identify this behavior, the web browser instruments all dialog messages that would normally require some user action to dismiss, including alerts, input boxes, and onbeforeunload events. When a dialog box occurs, the browser silently returns from the event, logging the text embedded in the dialog. If a return value is expected such as with an input box, the browser provides a random string as a response. The browser saves

as features the number of dialogs that occur, the text of the dialogs, and the presence of an onbeforeunload event.

Pop-up Windows. As with pop-up dialogs, pop-up windows are a common feature of spam. Whenever a pop-up window occurs, the browser allows the window to open, instrumenting the new page to collect all the same features as if the URL had originated from the dispatch queue. It records the parent URL that spawned the pop-up window, along with whether the page was launched via JavaScript or a plugin. After all windows have finished loading (or upon a timeout), the browser saves the total number of pop-up windows spawned and the features of each window and associates them with the parent URL.

Plugins. Previous reports have shown that spammers abuse plugins as a means to redirect victims to a final landing page [26], [29]. To capture such plugin behavior, our browser monitors all plugins instantiated by a page, the application type of each plugin (e.g., Java, Flash), and finally whether a plugin makes any request to the browser that leads to an outgoing HTTP request, causes a page to redirect, or launches a new window.

HTTP Headers. The HTTP headers that result as the browser loads a landing page provide a final source of information. Header data offers insight into the servers, languages, and versions of spam hosts, in addition to cookie values and custom header fields. We ignore HTTP fields and values associated with timestamps to remove any bias that results from crawling at particular times.

3.2. DNS Resolver

While spammers rapidly work through individual domain names during the course of a campaign, they often reuse their underlying hosting infrastructure for significant periods [17], [18], [20], [23]. To capture this information, once the web browser finishes processing a page, the crawler instance manager forwards the initial, final, and redirect URLs to a DNS resolver. For each URL, the resolver collects hostnames, nameservers, mailservers, and IP addresses associated with each domain. In addition, we examine whether a reverse lookup of the IP addresses reported match the domain they originated from. Each of these features provides a means for potentially identifying common hosting infrastructure across spam.

3.3. IP Address Analysis

Geolocation and routing information can provide a means for identifying portions of the Internet with a higher prevalence of spam [17]. To extract these features, we subject each IP address identified by the DNS resolver to further analysis in order to extract geolocation and routing data. This includes identifying the city, country, ASN, and BGP prefix associated with each address.

3.4. Proxy and Whitelist

To reduce network delay, Monarch proxies all outgoing network requests from a crawling instance through a single cache containing previous HTTP and DNS results. In addition, we employ a whitelist of known good domains and refrain from crawling them further if they appear during a redirect chain as a top-level window; their presence in IFrames or pop-up windows does not halt the surrounding collection process. Whitelists require manual construction and include trusted, high-frequency domains that do not support arbitrary user content. Our current whitelist contains 200 domains, examples of which include *nytimes.com*, *flickr.com*, and *youtube.com*. Whitelisted content accounts for 32% of URLs visited by our crawlers. The remaining content falls into a long tail distribution of random hostnames, 67% of which appear once and 95% of which appear at most 10 times in our system. While we could expand the whitelist, in practice this proves unnecessary and provides little performance improvement.

3.5. Feature Extraction

In preparation for classification, we transform the unprocessed features gathered during feature collection into a meaningful feature vector. We first canonicalize URLs to remove obfuscation, domain capitalization, and text encoding. Obfuscation includes presenting IP addresses in hex or octet format, or embedding path-traversal operations in the URL path. By reducing URLs to a canonical form, we can assure that common URL features match consistently across occurrences and cannot be masked by lexical transformations that would otherwise result in the same browser behavior. To compensate for potential lost information, we also include a boolean feature reflecting the presence of an obfuscated URL.

Once canonicalized, we split a URL into its basic components of domain, path, and query parameters, each of which we tokenize by splitting on non-alphanumeric characters. We apply a similar process to HTML and any text strings such as HTTP headers, where we tokenize the text corpus into individual terms and treat them as an unsorted bag of words. We then convert the results of tokenization into a binary feature vector, with a flag set for each term present. Rather than obscuring the origin of each token, we construct separate feature groups to indicate that a feature appeared in a redirect versus HTML content. Given the potential for millions of features, we represent feature vectors as sparse hash maps, which only indicate the presence of a feature. Finally, we provide these sparse maps to the classifier for training and decision making.

4. Distributed Classifier Design

For Monarch, we want a classifier that we can train quickly over large sets of data. Our first design decision along these lines was to use linear classification, where the output classification model is a weight vector $\vec{w}$ that describes

a hyperplane that separates data points placed in a high-dimensional space. We choose linear classification because of its simplicity, scalability and interpretability — for these reasons, linear classifiers have become common-place for web service providers interested in large-scale anti-phishing and anti-spam classification [24], [30].

In addition to quick training, we want the classifier to fit in memory. With these goals in mind, we design our training algorithm as a parallel online learner with regularization to yield a sparse weight vector. In particular, we combine the strategies of iterative parameter mixing [31] and subgradient L1-regularization [32]. Although more sophisticated algorithms exist that could yield higher accuracy classifiers at the expense of more training time, we favor a design that can yield favorable classification accuracies with less training time.

4.1. Notation

The problem of identifying spam URLs is an instance of binary classification. For a given URL, the data point $\vec{x} \in \mathbb{R}^d$ represents its feature vector in a high-dimensional space with d features. Because $\vec{x}$ is sparse (typically 1,000–1,500 nonzero entries out of an extremely large feature space of $d > 10^7$), we represent the feature vector as a hash map. We label the data point with an accompanying class label $y \in \{-1, +1\}$. $y = -1$ represents a non-spam site, and a $y = +1$ represents a malicious site.

To predict the class label of a previously unseen example $\vec{x}$ (representing the URL's feature vector), we train a linear classifier characterized by weight vector $\vec{w}$ trained offline on a labeled set of training data. During testing or deployment, we compute a predicted label as the sign of the dot product between the weight vector and the example: $\hat{y} = \text{sign}(\vec{x} \cdot \vec{w})$. If the predicted label $\hat{y} = +1$ but the actual label $y = -1$, then the error is a false positive. If $\hat{y} = -1$ but $y = +1$, then the error is a false negative.

4.2. Logistic Regression with L1-regularization

For training the weight vector $\vec{w}$, we use logistic regression (LR) with L1-regularization [33]. Given a set of n labeled training points $\{(\vec{x}_i, y_i)\}_{i=1}^n$, the goal of the training process is to find $\vec{w}$ that minimizes the following objective function:

$$f(\vec{w}) = \sum_{i=1}^{n} \log(1 + \exp[-y_i(\vec{x}_i \cdot \vec{w}_i)]) + \lambda \|\vec{w}\|_1. \qquad (1)$$

The first component of the objective constitutes the log-likelihood of the training data as a function of the weight vector—it is an upper bound on the number of mistakes made during training, and solving this proxy objective is a tractable alternative to directly minimizing the training error. For a single example $(\vec{x}_i, y_i)$, we can minimize the value of $\log(1 + \exp[-y_i(\vec{x}_i \cdot \vec{w}_i)])$ if the classification margin $y_i(\vec{x}_i \cdot \vec{w}_i)$ is a large positive value. (The margin is proportional to the distance between $\vec{x}_i$ and the classifier's decision boundary—a positive value means it is on the correct side of the boundary, and a negative value means it is on the incorrect side.) Thus, a solution that minimizes $f(\vec{w})$ would ideally yield a positive classification margin for as many examples as possible.

The second component is the regularization—it adds a penalty to the objective function for values where the L1 norm of $\vec{w}$ is large ($\|\vec{w}\|_1 = \sum_{j=1}^{d} w_j$). L1-regularization tends to yield sparse weight vectors—where there are relatively few nonzero feature weights. This is useful for applications that may be memory-constrained and require sparse solutions. (By contrast, using the L2 norm, another popular form of regularization, would yield solutions whose weights have small magnitudes but that tend to be non-sparse.) The parameter λ governs the amount of regularization: a higher λ gives the second component of Equation 1 more weight relative to the first component and will yield a more sparse solution.

Many optimization strategies exist for minimizing the objective in Equation 1. We had particular interest in a strategy amenable to learning over large-scale data sets in a short amount of time. We settled on a combination of recently-developed distributed learning and regularization techniques, which we describe in the next section.

Algorithm 1 Distributed LR with L1-regularization

Input: Data $\mathbf{D}$ with m shards
Parameters: λ (regularization factor), I (no. of iterations)
Initialize: $\vec{w} = \vec{0}$
for $i = 1$ to I **do**
  (gradient) $\vec{g}^{(j)} = \text{LRsgd}(\vec{w}, \mathbf{D}_j)$ for $j = 1..m$
  (average) $\vec{w} = \vec{w} - \frac{1}{m}\sum_{j=1}^{m} \vec{g}^{(j)}$
  (shrink) $w_\alpha = \text{sign}(w_\alpha) \cdot \max(0, |w_\alpha| - \lambda)$ for $\alpha = 1..d$
end for

Algorithm 2 Stochastic gradient descent for LR (LRsgd)

Input: $\vec{w}$ (weight vector), $\mathbf{D}_j$ (data shard)
Parameters: η (learning rate)
Initialize: $\vec{g}_0 = \vec{w}$
for $t = 1$ to $|\mathbf{D}_j|$ **do**
  Get data point $(\mathbf{x}_t, y_t)$ from $\mathbf{D}_j$
  Compute margin $z = y_t(\vec{x}_t \cdot (\vec{w} - \vec{g}_{t-1}))$
  Compute partial gradient $\vec{h} = y_t([1 + e^{-z}]^{-1} - 1)\vec{x}_t$
  $\vec{g}_t = \vec{g}_{t-1} + \eta\vec{h}$
end for
Return: $\vec{g}_{|\mathbf{D}_j|}$

4.3. Training Algorithm

We first divide the training data into m shards (which occurs by default storing data on certain distributed file systems such as the Hadoop Distributed File System [34]). Then, we distribute the initial model weight vector $\vec{w}$ to the m shards for training by stochastic gradient descent (Algorithm 1, "gradient" step).

Within each shard, we update the weight vector using

a stochastic gradient descent for logistic regression (Algorithm 2). We update the weight vector one example at a time as we read through the shard's data (this is also known as online learning). Under the constraint where we can only read each data item once within a shard, updating the model incrementally after every example typically has good convergence properties. As a performance optimization, we return the sum of the partial gradients rather than the updated weight vector itself.

After the m shards update their version of the weight vector, we collect the partial gradients $\vec{g}^{(1)}..\vec{g}^{(m)}$ and average them (Algorithm 1, "average" steps). Then, we perform L1-regularization (Algorithm 1, "shrink" step) on the averaged weight vector using a truncation function with threshold λ — this only applies to feature weights corresponding to binary features. In particular, all feature weights w_i with magnitude less than or equal to λ are set to 0, and all other weights have their magnitudes reduced by λ. This procedure reduces the number of nonzero weight vector entries, allowing the resulting weight vector to occupy less memory. Because there are fewer real-valued features (about 100) than binary features (about 10^7), we do *not* regularize the feature weights corresponding to real-valued features.

After the shrinkage step, we distribute the new weight vector $\vec{w}$ to the m shards again to continue the training process. The process repeats itself for I iterations.

A number of practical issues arise in getting the distributed logistic regression to scale to large-scale data. We describe how we implement our classifier in Section 5.4.

4.4. Data Set and Ground Truth

Our data set for training and testing the classifier consists of three sources: URLs captured by spam traps operated by major email providers, blacklisted URLs appearing on Twitter, and non-spam URLs appearing on Twitter that are used to represent a non-spam data sample. In total, we use Monarch's feature collection infrastructure over the course of two months to crawl 1.25 million spam email URLs, roughly 567,000 blacklisted Twitter URLs, and over 9 million non-spam Twitter URLs. Due to blacklist delay, generating our spam set of Twitter URLs requires retroactively checking all our Twitter URLs against 5 blacklists: Google Safebrowsing, SURBL, URIBL, Anti-Phishing Work Group (APWG), and Phishtank. If at any point after a URL is posted to Twitter its landing page, any of its redirects, frame URLs, or any of its source URLs become blacklisted, we treat the sample as spam. A breakdown of the categories of spam identified on Twitter can be seen in Table 2; 36% of blacklisted URLs were flagged as scams, 60% as phishing, and 4% as malware. A breakdown for email categories is not available, but the sample is known to contain scams, phishing, and malware.

In general, we lack comprehensive ground truth, which complicates our overall assessment of Monarch's performance. We may misclassify some true spam URLs as nonspam given absence of the URL in our spam-trap and blacklist feeds.

Blacklist	Detected URLs
Anti-Phishing Work Group	350,577
Google Safebrowsing (Phishing)[1]	12
Google Safebrowsing (Malware)[1]	22,600
Phishtank	46,203
SURBL (Scams)	51,263
SURBL (Malware, Phishing)	7,658
URIBL (Scams)	189,047
Total Samples	**667,360**
Total Unique	**567,784**

TABLE 2: Blacklist results for URLs appearing on Twitter that were flagged as spam.

Thus, we may somewhat underestimate false negatives (spam that slips through) seen in live operation, and overestimate false positives (legitimate URLs tagged as spam). In practice, building a training set of spam and non-spam samples remains a challenge for Monarch, requiring either user reports or spam traps operated by the web services seeking protection. However, for the purposes of evaluating Monarch's effectiveness at identifying spam, blacklists and email spam traps provide a suitable source of ground truth.

5. Implementation

We implement each of the four components of Monarch as independent systems operating on Amazon Web Services (AWS) cloud infrastructure. For exact specifications of the hardware used for our system, we refer readers to the AWS EC2 instance documentation [35].

5.1. URL Aggregation

URL aggregation and parsing is written in Scala and executes on a single EC2 Extra Large instance running Ubuntu Linux 10.04. The aggregation phase for Twitter URLs parses tweets from the Twitter Streaming API [36] and extracts URLs from the tweet text. The email spam URLs we process are provided to us post processing, and require no additional parsing. We place incoming URLs into a Kestrel queue [37] that keeps the most recent 300,000 URLs from the Twitter stream and email URLs to supply feature collection with a steady workload of fresh URLs. A full-fledged implementation of our system would require that the queue keeps every submitted URL, but for the purposes of evaluating Monarch, we only need enough URLs to scale to the system's throughput and to generate large data sets for classification.

5.2. Feature Collection

As previously framed, feature collection consists of four components: a web browser, DNS resolver, IP address analysis, and a monitor to handle message passing and aggregate

1. Twitter uses Google's Safebrowsing API to filter URLs appearing in tweets. URLs in our data set were either obfuscated to prevent detection, or were not present in the blacklist at the time of posting.

results. Feature collection runs in parallel on 20 EC2 High-CPU instances each running Ubuntu Linux 10.04 and executing 6 browsers, DNS resolvers, and IP analyzers each. For web browsing we rely on Firefox 4.0b4 augmented with a custom extension written in a combination of XML and JavaScript to tap into Firefox's API [38] which exposes browser-based events. We collect plugin-related events not exposed to the API by instrumenting Firefox's NPAPI [39] with hooks to interpose on all message passing between plugins and the browser. If a URL takes more than 30 seconds to load, we enforce a timeout to prevent delaying classification for other URLs. DNS resolution occurs over Linux's native `host` command, while geolocation and route lookups use the MaxMind GeoIP library [40] and Route Views [41] data respectively. A monitor written in Python aggregates the results from each of these services, generating its output as JSON text files stored in AWS S3.

5.3. Feature Extraction

Feature extraction is tightly coupled with the classification and training phase and does not run on separate hardware. Until the extraction phase, we store features in raw JSON format as key-value pairs. During extraction, we load the JSON content into a Scala framework, transform each into meaningful binary and real-valued features, and produce a sparse hash map stored in memory.

5.4. Classifier

Before training begins, we copy the raw feature data from Amazon S3 to a Hadoop Distributed File System (HDFS) [34] residing on the 50-node cluster of Amazon EC2 Double-Extra Large instances. Files in HDFS are automatically stored in shards of 128 MB, and we use this pre-existing partitioning (as required in the "Input" line of Algorithm 1). Within each shard, we randomize the order of the positive and negative examples—this gives the stochastic gradient descent in Algorithm 2 (which incrementally computes its partial gradient) better convergence rates compared to the situation of processing a long, contiguous block of positive examples followed by a long, contiguous block of negative examples.

We implement Algorithms 1 and 2 using Spark, a distributed computation framework that provides fault-tolerant distributed collections [42]. Spark provides map-reduce and shuffle operations, allow us to cache the data in memory across the cluster between iterations. We take advantage of these capabilities to construct an efficient distributed learner.

The first step of the training implementation is to normalize the real-valued features. In particular, we project real values to the $[0, 1]$ interval—doing so ensures that real-valued features do not dominate binary features unduly (a common practice in classification). We perform a map-reduce operation to compute the ranges (max/min values) of each real-valued feature. Then, we broadcast the ranges to the slave nodes. The slaves read the raw JSON data from HDFS, perform feature extraction to convert JSON strings into feature hash maps, and use the ranges to complete the normalization of the data vectors. At this point, the data is ready for training.

For the "gradient" step in Algorithm 1, we distribute m tasks to the slaves, whose job is to map the m shards to partial gradients. The slaves then compute the partial gradients for their respective shards using Algorithm 2.

Because the number of features is quite large, we want to avoid running Algorithm 1's "average" and "shrink" steps entirely on the master—the amount of master memory available for storing the weight vector $\vec{w}$ constitutes a resource bottleneck we must tend to.[2] Thus, we must avoid aggregating all of the partial gradients at the master at once and find an alternate implementation that exploits the cluster's parallelism.

To achieve this, we partition and shuffle the partial gradients across the cluster so that each slave is responsible for a computing the "average" and "shrink" steps on a disjoint subset of the feature space. We split each gradient into P partitions (not to be confused with the initial m data shards). Specifically, we hash each feature to an integer key value from 1 to P. Then, we shuffle the data across the cluster to allow the slave node responsible for feature partition $p \in \{1..P\}$ to collect its partial gradients. At this point the slave responsible for partition p performs the "average" and "shrink" steps. When these computations finish, the master collects the P partitions of the weight vector (which will have a smaller memory footprint than before shrinking) and joins them into the final weight vector $\vec{w}$ for that iteration.

6. Evaluation

In this section we evaluate the accuracy of our classifier and its run-time performance. Our results show that we can identify web service spam with 90.78% accuracy (0.87% false positives), with a median feature collection and classification time of 5.54 seconds. Surprisingly, we find little overlap between email and tweet spam features, requiring our classifier to learn two distinct sets of rules. We explore the underlying distinctions between email and tweet spam and observe that email is marked by short lived campaigns with quickly changing domains, while Twitter spam is relatively static during our two month-long analysis. Lastly, we examine our data set to illuminate properties of spam infrastructure including the abuse of popular web hosting and URL shorteners.

6.1. Classifier Performance

We train our classifier using data sampled from 1.2 million email spam URLs, 567,000 blacklisted tweet URLs, and 9 million non-spam URLs. In all experiments, we use the following parameters for training: we set the number of iterations to $I = 100$, the learning rate to $\eta = 1$, and the regularization factor to $\lambda = \frac{10\eta}{m}$ (where m is the number of data shards).

2. In our application, the slaves are able to compute the partial gradient over their respective shards without memory exhaustion. However, if the partial gradient computation were to bottleneck the slave in the future, we would have to add a regularization step directly to Algorithm 2.

Training Ratio	Accuracy	FP	FN
1:1	94.14%	4.23%	7.50%
4:1	**90.78%**	**0.87%**	**17.60%**
10:1	86.61%	0.29%	26.54%

TABLE 3: Results for training on data with different non-spam to spam ratios. We adopt a 4:1 ratio for classification because of its low false positives and reasonable false negatives.

Overall Accuracy. In order to avoid mistaking benign URLs as spam, we tune our classifier to emphasize low false positives and maintain a reasonable detection rate. We use a technique from Zadrozny et al. [43] to adjust the ratio of non-spam to spam samples in training to tailor false positive rates. We consider non-spam to spam ratios of 1:1, 4:1, and 10:1, where a larger ratio indicates a stronger penalty for false positives. Using 500,000 spam and non-spam samples each, we perform 5-fold validation and randomly subsample within a fold to achieve the required training ratio (removing spam examples to increase a fold's non-spam ratio), while testing always occurs on a sample made up of equal parts spam and non-spam. To ensure that experiments over different ratios use the same amount of training data, we constrain the training set size to 400,000 examples.

Table 3 shows the results of our tuning. We achieve lower levels of false positives as we apply stronger penalties, but at the cost of increased false negatives. We ultimately chose a 4:1 ratio in training our classifier to achieve 0.87% false positives and 90.78% overall accuracy. This choice strikes a balance between preventing benign URLs from being blocked, but at the same time limits the amount of spam that slips past classification. For the remainder of this evaluation, we execute all of our experiments at a 4:1 ratio.

To put Monarch's false positive rate in perspective, we provide a comparison to the performance of mainstream blacklists. Previous studies have shown that blacklist false positives range between 0.5–26.9%, while the rate of false negatives ranges between 40.2–98.1% [11]. Errors result from a lack of comprehensive spam traps and from low volumes of duplicate spam across all traps [44]. These same performance flaws affect the quality of our ground truth, which may skew our estimated false positive rate.

For web services with strict requirements on false positives beyond what Monarch can guarantee, a second tier of heavier-weight verification can be employed for URLs flagged by Monarch as spam. Operation can amortize the expense of this verification by the relative infrequency of false positives. Development of such a tool remains for future work.

Accuracy of Individual Components. Classification relies on a broad range of feature categories that each affect the overall accuracy of our system. A breakdown of the features used for classification before and after regularization can be found in Table 4. From nearly 50 million features we regularize down to 98,900 features, roughly half of which are each biased towards spam and non-spam. We do not include JavaScript pop-ups or plugin related events, as we found these on a negligible number of pages.

Feature Type	Unfiltered	Filtered	Non-spam	Spam
HTML terms	20,394,604	50,288	22,083	28,205
Source URLs	9,017,785	15,372	6,782	8,590
Page Links	5,793,359	10,659	4,884	5,775
HTTP Headers	8,850,217	9,019	3,597	5,422
DNS records	1,152,334	5,375	2,534	2,841
Redirects	2,040,576	4,240	2,097	2,143
Frame URLs	1,667,946	2,458	1,107	1,351
Initial/Final URL	1,032,125	872	409	463
Geolocation	5,022	265	116	149
AS/Routing	6,723	352	169	183
All feature types	**49,960,691**	**98,900**	**43,778**	**55,122**

TABLE 4: Breakdown of features used for classification before and after regularization.

Feature Type	Accuracy	FP	FN
Source URLs	89.74%	1.17%	19.38%
HTTP Headers	85.37%	1.23%	28.07%
HTML Content	85.32%	1.36%	28.04%
Initial URL	84.01%	1.14%	30.88%
Final URL	83.59%	2.34%	30.53%
IP (Geo/ASN)	81.52%	2.33%	34.66%
Page Links	75.72%	15.46%	37.68%
Redirects	71.93%	0.85%	55.37%
DNS	72.40%	25.77%	29.44%
Frame URLs	60.17%	0.33%	79.45%

TABLE 5: Accuracy of classifier when trained on a single type of feature. Sources, headers, and HTML content provide the best individual performance, while frame URLs and DNS data perform the worst.

To understand the most influential features in our system, we train a classifier exclusively on each feature category. For this experiment, we use the data set from the previous section, applying 10-fold validation with training data at a 4:1 non-spam to spam ratio and the testing set again at a 1:1 ratio. Any feature category with an accuracy above 50% is considered better than a classifier that naively guesses the majority population. The results of per-feature category training are shown in Table 5. Source URLs, which is an amalgamation of every URL requested by the browser as a page is constructed, provides the best overall performance. Had our classifier relied exclusively on initial URLs or final landing page URLs, accuracy would be 7% lower and false negatives 10% higher. Surprisingly, DNS and redirect features do not perform well on their own, each achieving approximately 72% accuracy. The combination of all of these features lowers the false positive rate while maintaining high accuracy.

Accuracy Over Time. Because criminals introduce new malicious websites on a continual basis, we want to determine how often we need to retrain our classifier and how long it takes for the classifier to become out of date. To answer these questions, we evaluate the accuracy of our classifier over a 20 day period where we had continuous spam and non-spam samples. We train using two different training regimens: (1) training the classifier once over four days' worth of data, then keeping the same classification model for the rest of the experiment; (2) retraining the classifier every four days, then testing the

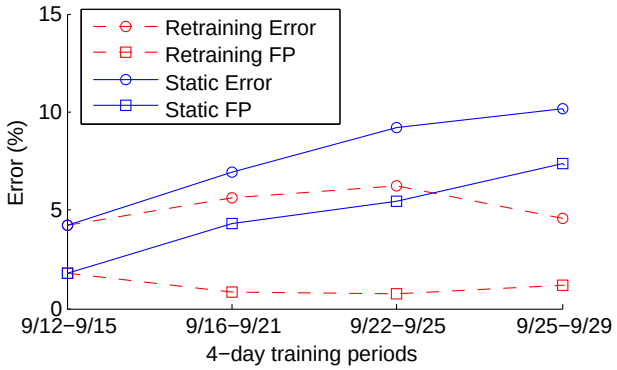

Fig. 3: Performance of classifier over time. Regular retraining is required to guarantee the best accuracy, else error slowly increases.

Training Set	Testing Set	Accuracy	FP	FN
Tweet spam	Tweet spam	94.01%	1.92%	22.25%
Tweet spam	Email spam	80.78%	1.92%	88.14%
Email spam	Tweet spam	79.78%	0.55%	98.89%
Email spam	Email spam	98.64%	0.58%	4.47%

TABLE 6: Effects of training and testing on matching and mismatching data sets. Email and tweet spam are largely independent in their underlying features, resulting in low cross classification accuracy.

Training Method	Accuracy	FP	FN
With Tweet Features	94.15%	1.81%	22.11%
Without Tweet Features	94.16%	1.95%	21.38%

TABLE 7: Effects of including contextual Twitter information. Omitting account and tweet properties from classification has no statistically significant effect on accuracy (the error rates are within one standard deviation of each another).

model on the subsequent four days of data. The data for each four-day window consists of 100,000 examples sampled at a 4:1 non-spam to spam ratio. We repeat this experiment four times by resampling each window's data, and take the average result.

Figure 3 shows the results for our time-sensitive evaluations. The error of the statically trained classifier gradually increases over time, whereas the classifier retrained daily maintains roughly constant accuracy. This indicates that in a deployment of Monarch, we will need to retrain the classifier on a continual basis. We explore the temporal nature of features that cause this behavior further in Section 6.3.

Training Across Input Sources. One of the primary challenges of training a classifier is obtaining labeled spam samples. Consequently, if a single labeled data set generalized to all web services, it would alleviate the problem of each web service being required to obtain their own spam samples. For instance, a great deal of time and effort could be saved if spam caught by passive email spam traps were applicable to Twitter where we currently are forced to crawl every link and retroactively blacklist spam URLs. However, spam targeting one web service is not guaranteed to be representative of spam targeting all web services. To this end we ask: how well can an email-trained classifier perform on Twitter data? How well can a Twitter-trained classifier perform on email data?

Table 6 displays the results of an experiment where we train our classifier on matching and mismatched data sources. We construct a 5-fold data set containing 400,000 non-spam samples and 100,000 tweet spam samples. Then, we copy the 5 folds but replace the 100,000 tweet spam samples with 100,000 email spam examples. We perform 5-fold cross validation to obtain classification rates. For a given testing fold, we test on both the tweet spam and email spam version of the fold (the non-spam samples remain the same in both version to ensure comparable results with respect to false positives).

Using a mixture of Twitter spam and non-spam samples, we are able to achieve 94% accuracy, but let 22% of spam tweets slip past our classifier. This same training regimen utterly fails on email, resulting in 88% of email spam going uncaught. These results are mirrored on a mixed data set of email spam and non-spam samples. We can achieve an accuracy of 98.64% with 4.47% false negatives when we train a classifier to exclusively find email spam. When we apply this same classifier to a testing set of Twitter spam, 98% of spam samples go uncaught.

These results highlight a fundamental challenge of spam filtering. Within the spam ecosystem, there are a variety of actors that each execute campaigns unique to individual web services. While Monarch's infrastructure generalizes to any web service, training data is not guaranteed to do the same. We require individual labeled data sets from each service in order to provide the best performance. A second unexpected result is the difficulty of identifying tweet spam compared to email spam. On matched training and testing sets, email spam classification achieves half the false negatives of tweet spam classification and a fifth of the false positives. We explore the underlying reason for this discrepancy in Section 6.3.

Context vs. Context Free Training. Because spam URLs can appear on different web services such as email, social networks, blogs, and forums, the question arises whether using context-aware features can improve classification accuracy at the cost of generalizability. To investigate this issue, we compare the error rate of classifying Twitter spam URLs (we exclude email spam) with and without account-based features. These features include account creation time, a tokenized version of tweet text, a tokenized version of an account's profile description, the number of friends and followers an account has, the number of posts made by an account, a tokenized screen name, the account's unique Twitter ID, the application used to access Twitter (e.g., web, Twitter's API, or a third-party application), hashtags present in the tweet, and "mentions" present in the tweet. Comprehensive historical data such as the ratio of URLs to posts is unavailable.

We perform 5-fold cross validation over a data set containing 400,000 non-spam samples and 100,000 tweet spam samples. The results of the experiment are shown in Table 7.

Even if Twitter account features are included, accuracy is statistically identical to training without these features. This contrasts with previous results that rely on account-based features to identify (fraudulent) spam accounts [12]–[14], but agrees with recent studies that have shown compromised accounts are the major distributors of spam [5], [11] which would render account-based features obsolete.

While this result is not guaranteed to generalize to all web services, we have demonstrated that strong performance for filtering email and Twitter spam is achievable without any requirement of revealing personally identifiable information. Omitting contextual information also holds promise for identifying web spam campaigns that cross web service boundaries without significant loss of accuracy due to disparate contextual information.

Component	Median Run Time (seconds)
URL aggregation	0.005
Feature collection	5.46
Feature extraction	0.074
Classification	0.002
Total	**5.54**

TABLE 8: Breakdown of the time spent processing a single URL.

Component	AWS Infrastructure	Monthly Cost
URL aggregation	1 Extra Large	$178
Feature collection	20 High-CPU Medium	$882
Feature extraction	—	$0
Classification	50 Double Extra Large	$527
Storage	700GB on EBS	$70
Total		**$1,587**

TABLE 9: Breakdown for the cost spent for Monarch infrastructure. Feature extraction runs on the same infrastructure as classification.

6.2. Run Time Performance

In addition to Monarch's accuracy, its overall performance and cost to execute are important metrics. In this section we measure the latency, throughput, and the cost of Monarch, finding a modest deployment of our system can classify URLs with a median time of 5.54 seconds and a throughput of 638,000 URLs per day, at a monthly cost of $1,600 on cloud machinery.

Latency. We measure latency as the time delta from when we receive a tweet or email URL until Monarch returns a final decision. Table 8 shows a breakdown of processing time for a sample of 5,000 URLs. URL aggregation takes 5 ms to parse a URL from Twitter's API format (email requires no parsing) and to enqueue the URL. Feature collection represents the largest overhead in Monarch, accounting for a median run time 5.46 seconds. Within feature collection, crawling a URL in Firefox consumes 3.13 seconds, while queries for DNS, geolocation and routing require 2.33 seconds. The majority of the processing time in both cases occurs due to network delay, not execution overhead. The remaining 70ms are spent extracting features and summing weight vectors for a classification decision.

Given that Firefox browsing incurs the largest delay, we investigate whether our instrumentation of Firefox for feature collection negatively impacts load times. We compare our instrumented Firefox against an uninstrumented copy using a sample of 5,000 URLs on a system running Fedora Core 13 machine with a four core 2.8GHz Xeon processor with 8GB of memory. We find instrumentation adds 1.02% overhead, insignificant to the median time it takes Firefox to execute all outgoing network requests which cannot be reduced. Instrumentation overhead results from interposing on browser events and message passing between the browser and monitoring service, accounting on average 110KB of log files.

Throughput. We measure the throughput of Monarch for a small deployment consisting of 20 instances on Amazon's EC2 infrastructure for crawling and feature collection. The crawling and feature extraction execute on a high-CPU medium instance that has 1.7GB of memory and two cores (5 EC2 compute units), running a 32-bit version of Ubuntu Linux 10.04. Each instance runs 6 copies of the crawling and feature collection code. We determined that the high-CPU medium instances have the lowest dollar per crawler cost, which make them the most efficient choice for crawling. The number of crawlers that each instance can support depends on the memory and CPU the machine. Using this small deployment, we can process 638,000 URLs per day.

Training Time. For the experiments in Section 6.1, we trained over data sets of 400,000 examples (80 GB in JSON format). The training time for 100 iterations of the distributed logistic regression took 45 minutes. Although we do not fully explore the effects of different data sizes or algorithm parameters on training time, we note that the following factors can increase the training time: a higher number of iterations, a larger training set (both with respect to number of examples and total number of nonzero features), a smaller regularization factor λ (which increases the amount of data communicated throughout the cluster by decreasing the sparsity of the partial gradients and weight vectors), and a smaller number of cluster machines.

For example, if we wanted to train on a larger number of examples, we could lower the number of itertations and increase the regularization factor to limit the training time. Being aware of these tradeoffs can help practitioners who want to retrain the classifier daily.

Cost. Using our deployment of Monarch as a model, we provide a breakdown of the costs associated with running Monarch on AWS for a month long period, shown in Table 9. Each of our components executes on EC2 spot instances that have variable prices per hour according to demand, while storage has a fixed price. URL aggregation requires a single instance to execute, costing $178 per month. For a throughput of 638,000 URLs per day, 20 machines are required to constantly crawl URLs and collect features, costing $882 per month. Besides computing, we require storage as feature data

accumulates from crawlers. During a one month period, we collected 1TB worth of feature data, with a cost of $.10 per GB. However, for live execution of Monarch that excludes the requirement of log files for experimentation, we estimate only 700GB is necessary to accommodate daily re-training at a monthly cost of $70. We can discard all other data from the system after it makes a classification decision. Finally, daily classifier retraining requires a single hour of access to 50 Double-Extra Large instances, for a total of $527 per month. In summary, we estimate the costs of running a URL filtering service using Monarch with a throughput of 638,000 URLs per day to be approximately $1,600 per month. We can reduce this cost by limiting our use of cloud storage (switching from JSON to a compressed format), as well as by reducing the processing time per URL by means of better parallelism and code optimizations.

We estimate the cost of scaling Monarch to a large web service, using Twitter as an example. Twitter users send 90 million tweets per day, 25% (22.5 million) of which contain URLs [45]. After whitelisting, deploying Monarch at that scale requires a throughput of 15.3 million URLs per day. The URL aggregation component is already capable of processing incoming URLs at this capacity and requires no additional cost. The crawlers and storage scale linearly, requiring 470 instances for feature collection and approximately 15 TB of storage for a week's worth of data, costing $20,760 and $1,464 per month respectively. The classifier training cost remains $527 per month so long as we use the same size of training sample. Alternatively, we could reduce the number of training iterations or increase the regularization factor λ to train on more data, but keep training within one hour. This brings the total cost for filtering 15.3 million URLs per day to $22,751 per month.

6.3. Comparing Email and Tweet Spam

We compare email and tweet spam features used for classification and find little overlap between the two. Email spam consists of a diverse ecosystem of short-lived hosting infrastructure and campaigns, while Twitter is marked by longer lasting campaigns that push quite different content. We capture these distinctions by evaluating two properties: feature overlap between email and tweet spam and the persistence of features over time for both categories. Each experiment uses 900,000 samples aggregated from email spam, tweet spam, and non-spam, where we use non-spam as a baseline.

Overlap. We measure feature overlap as the log odds ratio that a feature appears in one population versus a second population. Specifically, we compute $|log(p_1q_2/p_2q_1)|$, where p_i is the likelihood of appearing in population i and $q_i = 1 - p_i$. A log odds ratio of 0 indicates a feature is equally likely to be found in two populations, while an infinite ratio indicates a feature is exclusive to one population. Figure 4 shows the results of the log odds test (with infinite ratios omitted). Surprisingly, 90% of email and tweet features *never overlap*. The lack of correlation between the two indicates that email spammers are entirely separate actors from Twitter spammers, each pushing their own campaigns on distinct infrastructure. Consequently, the classifier must learn two separate sets of rules to identify both spam types.

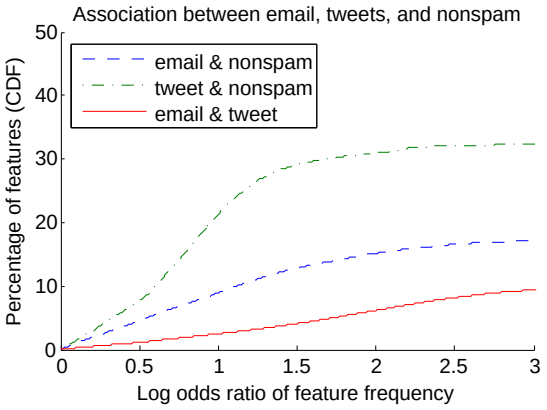

Fig. 4: Overlap of features. Email and Twitter spam share only 10% of features in common, indicating that email spammers and Twitter spammers are entirely separate actors.

Equally problematic, we find 32% of tweet spam features are shared with non-spam, highlighting the challenge of classifying Twitter spam. In particular, 41% of IP features associated with tweet spam are also found in nonspam, a result of shared redirects and hosting infrastructure. In contrast, only 16% of email spam IP features are found in non-spam, allowing a clearer distinction to be drawn between the two populations.

Persistence. We measure feature persistence as the time delta between the first and last date a feature appears in our data set, shown in Figure 5. Email spam is marked by much shorter lived features compared to tweet spam and non-spam samples. Notably, 77% of initial URL features appearing in email disappear after 15 days. The same is true for 60% of email DNS features, compared to just 30% of IP features associated with email spam hosting. Each of these results highlights the quick churn of domains used by email campaigns and the long lasting IP infrastructure controlled by email spammers. This same sophistication is unnecessary in Twitter, where there is no pressure to evade blacklists or spam filtering.

6.4. Spam Infrastructure

Email spam has seen much study towards understanding the infrastructure used to host spam content [18], [46]. From our feature collection, we identify two new properties of interest that help to understand spam infrastructure: redirect behavior used to lead victims to spam sites, and embedding spam content on benign pages.

Redirecting to spam. Both Twitter and email spammers use redirects to deliver victims to spam content. This mechanism

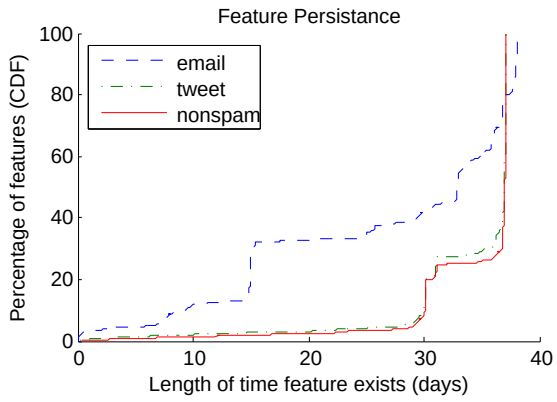

Fig. 5: Persistence of URL features. Email spam features are shorter lived compared to tweet spam, a result of short-lived campaigns and domain churn.

is dominated by tweet spam where 67% of spam URLs in our data set use redirects, with a median path length of 3. In contrast, only 20% of email spam URLs contain redirects, with a median path length of 2. Further distinctions between email and tweet spam behavior can be found in the abuse of public URL shorteners. Table 10 shows the top ten URL shortening services used for both email and tweet spam. The majority of email spam in our data set redirects through customized infrastructure hosted on arbitrary domains, while Twitter spammers readily abuse shortening services provided by bit.ly, Twitter, Google, and Facebook. Despite efforts by URL shorteners to block spam [8], [9], we find that widespread abuse remains prevalent.

Apart from the use of redirectors to mask initial URLs, we also examine domains that are commonly traversed as shortened URLs resolve to their final landing page. The top two destinations of URLs shortened by bit.ly are publicly available services provided by *google.com* and *blogspot.com*. Together, these two domains account for 24% of the spam first shortened by bit.ly. In the case of *google.com*, spam URLs embed their final landing page behind an arbitrary redirector operated by Google. This masks the final spam landing site from bit.ly, rendering blacklisting performed by the service obsolete. The second most common service, *blogspot.com*, is abused for free spam hosting rather than as a redirector. Each blog contains scam advertisements and other solicitations. By relying on Blogspot, spammers can evade domain-based blacklists that lack the necessary precision to block spam hosted alongside benign content.

Each of these are prime examples of web services currently being abused by spammers and serve as a strong motivation for the need of a system like Monarch.

Page content. Another phenomenon we frequently observe in Twitter spam is the blacklisting of content within a page. For the majority of sites, this is a web advertisement from a questionable source. We have observed popular news sites with non-spam content displaying ads that cause a variety

Domain	Email spam	Twitter spam
bit.ly	1%	41%
t.co	0%	4%
tinyurl.com	3%	4%
ow.ly	0%	4%
goo.gl	0%	3%
su.pr	0%	3%
fb.me	0%	2%
dlvr.it	0%	2%
os7.biz	0%	1%
is.gd	0%	1%

TABLE 10: Top 10 URL shortening services abused by spammers.

Feature Category	% Blacklisted	% Exclusive
Initial URL	16.60%	0.05%
Final URL	23.33%	2.62%
Top-level Window Redirect URL	34.25%	4.41%
Content Redirect URL	3.99%	1.35%
Frame Content URL	14.85%	6.87%
Link URLs	28.28%	7.03%
Source URLs	100%	42.51%

TABLE 11: Breakdown of the locations of blacklisted URLs. We mark a page as spam if it makes any outgoing request to a blacklisted URL.

of spam popups, sounds, and video to play. Table 11 shows a breakdown of the locations containing blacklisted URLs specifically for Twitter. The column labeled exclusive indicates the percent of URLs that can be blacklisted exclusively based on a URL in that location. For example, 0.05% of Twitter spam can be blacklisted using only an initial URL posted to the site. Since the category source URLs is a superset of all other URLs, 100% of pages can be blacklisted; however, looking exclusively at URLs which are not part of other categories, we find that 42.51% of source URLs lead to blacklisting. This indicates a page included an image, stylesheet, plugin, script, or dynamically retrieved content via JavaScript or a plugin that was blacklisted. These scenarios highlight the requirement of analyzing *all* of a webpages content to not overlook spam with dynamic page behavior or mash-up content that includes known spam domains.

7. Discussion

In this section we discuss potential evasive attacks against Monarch that result from running a centralized service. While we can train our system to identify spam and have shown the features we extract are applicable over time, classification exists in an adversarial environment. Attackers can tune features to fall below the spam classification threshold, modify content after classification, and block our crawler. We do not propose solutions to these attacks; instead, we leave to future work an in depth study of each attack and potential solutions.

Feature Evasion. When Monarch provides a web service with a classification decision, it also provides attackers with immediate feedback for whether their URLs are blocked. An attacker can use this feedback to tune URLs and content in an attempt to evade spam classification, as discussed in

previous studies [47]–[49], but not without consequences and limitations. The simplest changes an attacker can make are modifications to page content: HTML, links, and plugins. Known spam terms can be transformed into linguistically similar, but lexically distinct permutations to avoid detection, while links and plugins can be modified to imitate non-spam pages. Page behavior poses a more difficult challenge; by removing pop-up windows and alert prompts, a spammer potentially reduces the effectiveness of eliciting a response from victims. Finally, hosting infrastructure, redirects, and domains, while mutable, require a monetary expense for dynamism. We leave evaluating how susceptible our classification system is to evasion to future work, but note that email spam classification and intrusion prevention systems both exist in adversarial environments and maintain wide-spread adoption.

Time-based Evasion. In Monarch's current implementation, feature collection occurs at the time a URL is submitted to our system; URLs are not re-crawled over time unless they are resubmitted. This raises the potential for an attacker to change either page content or redirect to new content after a URL has been classified. For this attack to succeed, a URL's redirects and hosting infrastructure must appear benign during classification and allow subsequent modification. An attacker that simply masks his final landing page, but re-uses known hostile redirect infrastructure may still be identified by the classifier. Furthermore, static shorteners such as bit.ly cannot be used because the landing page cannot be changed after shortening. To circumvent both of these limitations, an attacker can rely on mutable content hosted on public infrastructure typically associated with non-spam pages, such as Blogspot, LiveJournal, and free web hosting. In this scenario, an attacker's blog contains non-spam content during classification and is subsequently modified to include spam content or a JavaScript redirect to a new hostile landing page.

Crawler Evasion. Rather than an attacker modifying content to evade classification, an adversary can alter HTTP and DNS behavior to prevent our crawler from ever reaching spam pages. Potential attacks include relying on browser user-agent detection or other forms of browser fingerprinting [50] to forward our crawler to non-hostile content and regular users to a hostile copies. Alternatively, the IP addresses of Monarch's crawlers can be learned by an attacker repeatedly posting URLs to our service and tracking the IPs of visitors. A list of crawler IP addresses can then be distributed as a blacklist, with attackers either blocking access or redirecting our crawlers to non-spam content.

8. Related Work

Web Service-specific Defenses. Of the threats facing web services, social networks in particular have garnered particular attention. In a recent study, Gao et al. showed that 10% of links posted to Facebook walls are spam while 97% of accounts participating in the campaigns are compromised users [4]. The same is true of Twitter, where at least 8% of links posted are spam and 86% of the accounts involved are compromised [5]. To counter this threat, a number of service-specific solutions have been proposed to classify spam accounts based on post frequency, the number of URLs an account posts, and the ability of an account to acquire friends [12]–[14]. Accuracy for these techniques varies between 70–99% for detecting spam accounts, with sample sizes ranging between 200–1500 spam accounts. However, many of the metrics used to detect spammers can be readily evaded. Spammers can randomize the template behind spam messages posted to social networks, post benign messages to skew the ratio of URLs to posts, befriend other spammers to mimic the average friend counts of regular users, and generate dormant spam accounts prior to the onset of a spam campaign to avoid heuristics targeting account age.

While Twitter and Facebook rely on similar account heuristics for identifying automated spammers [51], [52], these heuristics have limited utility in identifying compromised accounts and incur a delay between a fraudulent account's detection and creation to develop a history of (mis-)activity. In contrast, Monarch operates independently of account properties, generalizes to all web services that receive URL spam, and requires no monitoring window.

Characterizing Spam. Monarch builds on a large foundation of previous efforts to characterize spam properties. This includes the lexical characteristics of phishing URLs [16], the locality of email spammer IP addresses [17], hosting infrastructure and page layout that is shared across email spam campaigns [18], and the content of spam websites [27]. In turn, these properties have been used to develop techniques to detect algorithmically generated spam domains [21] and to identify spam domains based on nameserver and registration times [20]. We expand upon these properties, adding our own study of spam behavior including a comparison between email and tweet spam as well as services abused by each.

Detecting Scams, Phishing, and Malware. Detecting scams, phishing, and malware based on URL and page properties is by no means new. Particular attention has been paid to identifying phishing URLs, where a number of solutions rely on HTML forms, input fields, page links, URL features, and hosting properties for detection [15], [28], [53]. Malware, specifically drive-by-downloads, has also been the target of recent study, with most solutions relying on exposing sandboxed browsers to potentially malicious content [54], [55]. An exception is Wepawet, which relies on detecting anomalous arguments passed to plugins to prevent attacks [56]. Our own system generalizes to all forms of scams, phishing, and malware and allows for real-time URL submission by web services.

Of the closest works to our own, Ma et al. show that one can classify spam URLs based on lexical structure and underlying hosting infrastructure including DNS and WHOIS information [22], [23]. We employ these same metrics in our system, but crawl URLs to resolve redirect URLs that would otherwise obscure the final landing page and its hosting

infrastructure. A similar approach is taken by Wittaker et al. [24] for specifically classifying phishing pages. We expand upon their research and generalize Monarch to detect all forms of spam, adding features such as JavaScript behavior, redirect chains, and the presence of mashup content, while developing our own classification engine and collection infrastructure to fulfill real-time requirements.

Spam Filtering and Usability Challenges. In this paper, we particularly focus on providing accurate decisions for whether a URL directed to spam content. However, a second challenge remains for web services, as they must decide how to appropriately address spam content. Currently, Twitter and Facebook prevent messages containing known spam content from being posted [57], [58], while bit.ly implements a warning page that users must click past to access potentially harmful content [9]. Warnings provide users with an opportunity to bypass false positives, but burden users with making (un-)informed security decisions.

The effectiveness of warnings in the context of phishing sites was examined in several studies [59], [60], the results of which showed that unobtrusive warning messages are ineffective compared to modal dialogs and active, full-screen warnings. This work lead to a discussion of the best approach for educating users of security practices and making informed decisions [61]. While advances in usability are orthogonal to Monarch, web services relying on Monarch's decisions can take heed of these studies when determining the best mechanism for conveying the potential harm of a URL to users.

9. Conclusion

Monarch is a real-time system for filtering scam, phishing, and malware URLs as they are submitted to web services. We showed that while Monarch's architecture generalizes to many web services being targeted by URL spam, accurate classification hinges on having an intimate understanding of the spam campaigns abusing a service. In particular, we showed that email spam provides little insight into the properties of Twitter spammers, while the reverse is also true. We explored the distinctions between email and Twitter spam, including the overlap of spam features, the persistence of features over time, and the abuse of generic redirectors and public web hosting. We have demonstrated that a modest deployment of Monarch on cloud infrastructure can achieve a throughput of 638,000 URLs per day with an overall accuracy of 91% with 0.87% false positives. Each component of Monarch readily scales to the requirements of large web services. We estimated it would cost $22,751 a month to run a deployment of Monarch capable of processing 15 million URLs per day.

10. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0311808, 0832943, 0448452, 0842694, 0627511, 0842695, 0831501, 0433702, 0905631, CCF-0424422, and CNS-0509559. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This work is partially supported by the Office of Naval Research under MURI Grant No. N000140911081. This material is based upon work supported by the MURI program under AFOSR Grant No: FA9550-08-1-0352. This research is also supported by gifts from Sun Microsystems, Google, Microsoft, Amazon Web Services, Cisco Systems, Cloudera, eBay, Facebook, Fujitsu, Hewlett-Packard, Intel, Network Appliance, SAP, VMWare and Yahoo! and by matching funds from the State of California's MICRO program (grants 06-152 and 07-010), and the University of California Industry/University Cooperative Research Program (UC Discovery) grant COM07-10240.

We thank Matei Zaharia and John Duchi for discussions about the distributed learning implementation. We also thank our shepherd Ben Livshits and our anonymous reviewers for their input.

References

[1] G. Cluley, "This you????: Phishing attack hits twitter users." http://www.sophos.com/blogs/gc/g/2010/02/24/phishing-attack-hits-twitter-users/, 2010.

[2] E. Mills, "Facebook hit by phishing attacks for a second day," *CNET News*, 2009.

[3] John E. Dunn, "Zlob Malware Hijacks YouTube," 2007. http://www.pcworld.com/article/133232/zlob_malware_hijacks_youtube.html.

[4] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, and B. Zhao, "Detecting and characterizing social spam campaigns," in *Proceedings of the Internet Measurement Conference (IMC)*, 2010.

[5] C. Grier, K. Thomas, V. Paxson, and M. Zhang, "@spam: the underground on 140 characters or less," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2010.

[6] Kim Zetter, "Trick or Tweet? Malware Abundant in Twitter URLs," *Wired*, 2009.

[7] B. Stone, "Facebook Joins With McAfee to Clean Spam From Site," *New York Times*, 2010.

[8] HootSuite, "Kapow! HootSuite Fights the Evils of Phishing, Malware, and Spam," 2010. http://blog.hootsuite.com/hootsuite-fights-malware-phishing/.

[9] bit.ly, "Spam and Malware Protection," 2009. http://blog.bit.ly/post/138381844/spam-and-malware-protection.

[10] A. Ramachandran, N. Feamster, and S. Vempala, "Filtering spam with behavioral blacklisting," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[11] S. Sinha, M. Bailey, and F. Jahanian, "Shades of grey: On the effectiveness of reputation-based blacklists," in *3rd International Conference on Malicious and Unwanted Software*, 2008.

[12] F. Benevenuto, G. Magno, T. Rodrigues, and V. Almeida, "Detecting Spammers on Twitter," in *Proceedings of the Conference on Email and Anti-Spam (CEAS)*, 2010.

[13] K. Lee, J. Caverlee, and S. Webb, "Uncovering social spammers: social honeypots+ machine learning," in *Proceeding of the International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2010.

[14] G. Stringhini, C. Kruegel, and G. Vigna, "Detecting Spammers on Social Networks," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2010.

[15] C. Ludl, S. McAllister, E. Kirda, and C. Kruegel, "On the effectiveness of techniques to detect phishing sites," *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2007.

[16] D. McGrath and M. Gupta, "Behind phishing: an examination of phisher modi operandi," in *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, 2008.

[17] S. Venkataraman, S. Sen, O. Spatscheck, P. Haffner, and D. Song, "Exploiting network structure for proactive spam mitigation," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007.

[18] D. Anderson, C. Fleizach, S. Savage, and G. Voelker, "Spamscatter: Characterizing internet scam hosting infrastructure," in *USENIX Security*, 2007.

[19] A. Pitsillidis, K. Levchenko, C. Kreibich, C. Kanich, G. Voelker, V. Paxson, N. Weaver, and S. Savage, "Botnet Judo: Fighting spam with itself," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

[20] M. Felegyhazi, C. Kreibich, and V. Paxson, "On the potential of proactive domain blacklisting," in *Proceedings of the USENIX Conference on Large-scale Exploits and Emergent Threats*, April 2010.

[21] S. Yadav, A. Reddy, A. Reddy, and S. Ranjan, "Detecting Algorithmically Generated Malicious Domain Names," in *Proceedings of the Internet Measurement Conference (IMC)*, 2010.

[22] J. Ma, L. Saul, S. Savage, and G. Voelker, "Beyond blacklists: learning to detect malicious web sites from suspicious urls," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009.

[23] J. Ma, L. Saul, S. Savage, and G. Voelker, "Identifying suspicious URLs: an application of large-scale online learning," in *Proceedings of the 26th Annual International Conference on Machine Learning*, 2009.

[24] C. Whittaker, B. Ryner, and M. Nazif, "Large-Scale Automatic Classification of Phishing Pages," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

[25] K. Chellapilla and A. Maykov, "A taxonomy of JavaScript redirection spam," in *Proceedings of the 3rd International Workshop on Adversarial Information Retrieval on the Web*, 2007.

[26] Dan Goodin, "Scammers skirt spam shields with help from Adobe Flash," *The Register*, 2010. http://www.theregister.co.uk/2008/09/04/spammers_using_adobe_flash/.

[27] A. Ntoulas, M. Najork, M. Manasse, and D. Fetterly, "Detecting spam web pages through content analysis," in *Proceedings of the 15th International Conference on World Wide Web*, 2006.

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

[29] K. Thomas and D. M. Nicol, "The Koobface botnet and the rise of social malware," in *Proceedings of The 5th International Conference on Malicious and Unwanted Software (Malware 2010)*, 2010.

[30] K. Q. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature Hashing for Large Scale Multitask Learning," in *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 681–688, June 2009.

[31] R. McDonald, K. Hall, and G. Mann, "Distributed Training Strategies for the Structured Perceptron," in *Proceedings of the North American Association for Computing Linguistics (NAACL)*, (Los Angeles, CA), June 2010.

[32] J. Duchi and Y. Singer, "Efficient Online and Batch Learning Using Forward Backward Splitting," *Journal of Machine Learning Research*, vol. 10, pp. 2899–2934, Dec. 2009.

[33] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York, NY: Springer, 2009.

[34] Hadoop, "Hadoop Distributed File system." http://hadoop.apache.org/hdfs/, 2010.

[35] Amazon Web Services, "Amazon EC2 Instance Types," 2009. http://aws.amazon.com/ec2/instance-types/.

[36] Twitter, "Twitter API wiki." http://apiwiki.twitter.com/Twitter-API-Documentation, 2010.

[37] Twitter, "Building on open source." http://blog.twitter.com/2009/01/building-on-open-source.html, 2010.

[38] Mozilla, "API & Language References." https://addons.mozilla.org/en-US/developers/docs/reference, 2010.

[39] Mozilla, "Netscape plugin API." http://www.mozilla.org/projects/plugins/, 2004.

[40] MaxMind, "Resources for Developers." http://www.maxmind.com/app/api, 2010.

[41] Advanced Network Technology Center, "Univeristy of Oregon route views project." http://www.routeviews.org/, 2010.

[42] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2nd USENIX Conference on Hot topics in Cloud Computing*, (Boston, MA), June 2010.

[43] B. Zadrozny, J. Langford, and N. Abe, "Cost-Sensitive Learning by Cost-Proportionate Example Weighting," in *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, (Melbourne, FL), Nov. 2003.

[44] S. Sinha, M. Bailey, and F. Jahanian, "Improving spam blacklisting through dynamic thresholding and speculative aggregation," in *Proceedings of the 17th Annual Network & Distributed System Security Symposium*, 2010.

[45] L. Rao, "Twitter seeing 90 million tweets per day, 25 percent contain links." http://techcrunch.com/2010/09/14/twitter-seeing-90-million-tweets-per-day/, September 2010.

[46] T. Holz, C. Gorecki, F. Freiling, and K. Rieck, "Detection and mitigation of fast-flux service networks," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS08)*, 2008.

[47] N. Dalvi, P. Domingos, S. Mausam, and D. Verma, "Adversarial classification," in *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, 2004.

[48] D. Lowd and C. Meek, "Adversarial learning," in *Proceedings of the International Conference on Knowledge Discovery in Data Mining*, 2005.

[49] M. Barreno, B. Nelson, R. Sears, A. Joseph, and J. Tygar, "Can machine learning be secure?," in *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, 2006.

[50] P. Eckersley, "How Unique Is Your Web Browser?," in *Privacy Enhancing Technologies (PET)*, 2010.

[51] Twitter, "The twitter rules." http://support.twitter.com/entries/18311-the-twitter-rules, 2010.

[52] C. Ghiossi, "Explaining Facebook's Spam Prevention Systems." http://blog.facebook.com/blog.php?post=403200567130, 2010.

[53] S. Garera, N. Provos, M. Chew, and A. Rubin, "A framework for detection and measurement of phishing attacks," in *Proceedings of the 2007 ACM Workshop on Recurring Malcode*, 2007.

[54] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose, "All your iFRAMEs point to us," in *Proceedings of the 17th Usenix Security Symposium*, pp. 1–15, July 2008.

[55] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King, "Automated Web patrol with Strider HoneyMonkeys: Finding Web sites that exploit browser vulnerabilities," in *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS)*, February 2006.

[56] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious JavaScript code," in *Proceedings of the 19th International Conference on World Wide Web*, 2010.

[57] F-Secure, "Twitter now filtering malicious URLs." http://www.f-secure.com/weblog/archives/00001745.html, 2009.

[58] Facebook, "Explaining Facebook's spam prevention systems." http://blog.facebook.com/blog.php?post=403200567130, 2010.

[59] M. Wu, R. Miller, and S. Garfinkel, "Do security toolbars actually prevent phishing attacks?," in *Proceedings of the SIGCHI conference on Human Factors in computing systems*, 2006.

[60] S. Egelman, L. Cranor, and J. Hong, "You've been warned: an empirical study of the effectiveness of web browser phishing warnings," in *Proceeding of the Conference on Human Factors in Computing Systems*, 2008.

[61] C. Herley, "So long, and no thanks for the externalities: The rational rejection of security advice by users," in *Proceedings of the 2009 Workshop on New Security Paradigms Workshop*, 2009.