# Beating the Artificial Chaos: Fighting OSN Spam using Its Own Templates

Tiantian Zhu, Hongyu Gao, Yi Yang, Kai Bu, *Member, IEEE*, Yan Chen, *Senior Member, IEEE*, Doug Downey, Kathy Lee, and Alok Choudhary *Fellow, IEEE*

***Abstract*—Online social networks (OSNs) are extremely popular among Internet users. However, spam originating from friends and acquaintances not only reduces the joy of Internet surfing but also causes damage to less security-savvy users. Prior countermeasures combat OSN spam from different angles. Due to the diversity of spam, there is hardly any existing method that can independently detect the majority or most of OSN spam. In this paper, we empirically analyze the textual pattern of a large collection of OSN spam. An inspiring finding is that the majority (e.g., 76.4% in 2015) of the collected spam is generated with underlying templates. Based on the analysis, we propose *Tangram*, an OSN spam filtering system that performs online inspection on the stream of user-generated messages. Tangram extracts templates of spam detected by existing methods and then matching messages against the templates toward accurate and fast spam detection. It automatically divides OSN spam into segments and uses the segments to construct templates to filter future spam. Experimental results on Twitter and Facebook datasets show that Tangram is highly accurate and can rapidly generate templates to throttle newly emerged campaigns. Furthermore, we analyze the behavior of detected OSN spammers. We find a series of spammer properties—such as spamming accounts are created in bursts and a single active organization orchestrates more spam than all other spammers combined—that promise more comprehensive spam countermeasures.**

***Index Terms*—Online social networks, spam, spam campaigns.**

## I. Introduction

SPAMMERS started exploiting online social networks (OSNs) as soon as OSN became a hit [2]. OSN spam not only reduces the joy of Internet surfing but also causes damage to less security-savvy users. Researchers propose combating OSN spam from different angles, including mining the textual content [3], studying the redirection chains of embedded URLs [4], as well as classifying the URL landing pages [5]. Despite the development of countermeasures, spammers find their way to adapt and stick. Back to 2011, on Twitter, one of the most popular OSNs nowadays, more than 4% of collected tweets are spam [6], which has slipped through all the deployed defense mechanisms. While up to 2014, 5% of the entire Twitter's user base are spam bots [7]. Why could this still happen given the efforts to build various spam mitigation systems? We observe that a primary reason is the absence of clear understandings of what techniques the spammers are using to construct OSN spam and how the techniques evolve. Such missing piece of fundamental information is critical for exploring effective designs to throttle OSN spam.

Toward uncovering OSN spam generation techniques, we conduct a large-scale, consecutive measurement study (Section II). We find that the majority of spam is generated with underlying templates, which is consistent with prior email spam research [8]–[10]. Templates are valuable for spammers, because they let spammers control and customize the semantic meaning of generated messages to boost the conversion rate. OSN spammers have evolved to use more sophisticated templates that break the assumptions in prior email spam template generation research, making them ineffective for OSN spam.

In particular, our measurement results reveal three challenges that render template-based OSN spam hard to throttle.

**Absence of invariant substring in template.** Prior spam template generation research [10], [11] made a crucial assumption that an invariant substring is hard-coded in a template, so that every instantiation of the template contains such string. Unfortunately, an OSN spam template does not always contain any invariant substring.

**Prevalence of noise.** Spammers extensively add semantically unrelated noise words into spam messages. The presence of noise diversifies spam, and increases the difficulty to identify semantically meaningful text segments.

**Spam heterogeneity.** Spam instantiating different templates mixes with spam without any underlying templates. It is hard to obtain a training set with a single template in an online detection scenario.

In this paper, we propose Tangram system to combat OSN spam through effective spam-template generation. Tangram stands out among existing spam countermeasures because of three properties. First, Tangram directly tackles spam whereas many existing methods detect spammers instead of spam. Such methods are based on account activity and need long observation periods for the account features to accumulate [12]–[15]. Second, some other detection approaches are based on URL analysis, which inherently cannot detect spam without

This work is supported in part by grants NSF awards CCF-1029166, ACI-1144061, IIS-1343639, CCF-1409601 and CNS-1064595; DOE award DESC0007456; DARPA contract D11AP00268; Fundamental Research Funds for the Central Universities under Grant No. 2014QNA5012; National Science Foundation of China under Grant No. 61402404 and No. 61472359. A preliminary version of this manuscript has been published in the Proceedings of 2014 Annual Computer Security Applications Conference (ACSAC) [1]. (Corresponding Author: Kai Bu)

T. Zhu and K. Bu are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China (e-mail: {ttzhu, kaibu}@zju.edu.cn).

H. Gao is with Google Inc., Mountain View, CA 94043, USA (e-mail: gargoylehon@gmail.com).

Y. Yang, Y. Chen, D. Downey, K. Lee, and A. Choudhary are with Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208 USA (e-mail: yiyang2014@u.northwestern.edu, ychen@northwestern.edu, {ddowney, kathy.lee, choudhar}@eecs.northwestern.edu).

URLs [4], [5], [16]. Researchers have revealed significant amount of such spam [17]. The few existing methods that detect spam with or without URLs in real-time suffer from high false positive rates [3], [17], [18]. In contrast, Tangram is the first accurate online OSN spam detection system that detects spam with or without URLs. Third, what is more unique to Tangram is that it directly hits OSN spam's vital point—template-based spam that counts the most. Tangram extracts templates of spam detected by existing methods and then matches messages against the templates toward accurate and fast spam detection. Beyond spam detection, we further investigate the detected spammers to infer their strategy.

In summary, Tangram is highly accurate because of the following sweet spots.

**Embrace the absence of invariant substring.** We identify frequently appearing segments within messages and then locate equivalent segments among messages. Such segments are later assembled into spam templates for matching future spam.

**Mitigate the prevalence of noise.** We cast a sequence-labeling task to label each word in a given message as either "noise" or "non-noise". Only "non-noise" words yield templates.

**Break spam heterogeneity.** We pre-cluster spam and perform template generation within individual partitions. We also discard outlier messages in the partition.

**Build a double defense.** We mitigate spam without underlying templates using a supplementary module that detects spam with excessive semantically unrelated noise words. In addition, we can equip Tangram with multiple heterogeneous detection modules in practice.

## II. Motivation: Revealing Twitter Spam Template

In this section, we empirically analyze the textual patterns of Twitter spam as a first attempt to quantitatively reveal popular techniques that generate current OSN spam.

### A. Data Collection

We first collected a large dataset from Twitter that contained about 17 million public tweets generated by 4.2 million users. The tweets were generated between June 1, 2011 and July 21, 2011. We continuously downloaded popular Twitter Hashtags from the website *What the Trend* [19]. We then downloaded all public tweets that contained the Hashtags. Our data collection method was inevitably biased towards tweets containing popular Hashtags. Consequently, the spam tweets in our dataset were also biased towards spammers using hashtags. However, the numerical accountIDs in our dataset followed a uniform distribution (as shown in Figure 4 in Section V-A), suggesting the dataset was not overly biased towards specific account groups.

We revisited Twitter in March 2012 to label the collected tweets. For each account that posted tweets in the dataset, we crafted a special URL, using which we could access the account's personal profile on Twitter. We only checked if we were redirected to a page indicating that the account had been suspended. We found that 120,386 accounts were suspended. They posted 558,706 tweets, all of which were labeled as spam. There were 532,676 unique spam tweets,

TABLE I
Retrofitted sample spam from a template-based campaign.

| | | |
|---|---|---|
| Big Name A | an eye-catching action - | *URL* |
| Celebrity B | an eye-catching action - | *URL* |
| Big Name A | offensive content , look at this video | *URL* |
| Celebrity B | offensive content , look at this video | *URL* |
| RIP Celeb C | offensive content , look at this video | *URL* |

*Notes: We intentionally substituted likely offensive contents such as celebrity identities and sexuality in the example spam tweets. Interested readers are welcome to contact the authors for original datasets.*

TABLE II
Retrofitted sample spam from a paraphrase campaign.

| | |
|---|---|
| browsing statistics click Celebrity John | *URL* |
| interesting site on Celeb J | *URL* |
| the actual webapge ever Celeb J | *URL* |

showing that our collected spam had few duplicates. The other tweets were labeled as legitimate. The labeled spam tweets did not represent the spam that Twitter could already detect. Rather, their presence in our collected data meant that Twitter failed to detect them when they were generated [1]. We thus usually need several months delay to get ground-truth spam tweets after they are posted.

Using the same method, we collected three more relatively small tweet datasets generated during January 2012 (2 million from Jan 1, 2012 to Jan 6, 2012), October 2014 (9 million from Oct 1, 2014 to Oct 31, 2014), and January 2015 (3 million from Jan 1, 2015 to Jan 31, 2015), respectively containing 46,844, 20,977, and 6,372 spam tweets.

### B. Spam Categorization

Our goal is to study the textual pattern of spam. We use human intelligence to perceive the spam's semantic meaning. We first split spam into two categories: "semantically similar" and "semantically dissimilar". Semantically similar spam forms big clusters that share the same semantic meaning, whereas semantically dissimilar spam does not. We further divide the "semantically similar" spam into "*template-based*" and "*paraphrase*" spam. For example, a cluster with "This can be just" as cluster character contains "This can be just wonderful.", "This can be just awesome.", "This can be just stunning.", and so on, is obviously template-based. Another cluster with "are you invented" as cluster character contains messages such as "What a fantastic celebration. Do you think you're invented." and "How much of an fantastic celebration. Do you think you're welcome?" is obviously paraphrase. In addition, we identify a sub-category of semantically dissimilar spam that contains very little semantically meaningful textual content. These messages contain a long list of popular keywords and hashtags. We denote it as the "*no-content category*". Besides, we denote other semantically dissimilar spam that we have not systematically categorized as "*other spam*".

**Template-based Category** consists of the majority of spam tweets and exhibits patterns of underlying templates. The most prominent characteristic is that all the tweets in the same campaign can be uniformly disassembled into multiple components, so that the corresponding component serves the same semantic meaning in the sentence. In addition, these components obey the same ordering. It is common for two spam tweets to share common substrings, but the campaign

TABLE III
THE POPULARITY OF FOUR SPAM CATEGORIES IN JUNE/JULY, 2011, JANUARY, 2012, OCTOBER, 2014, AND JANUARY, 2015, RESPECTIVELY.

| Spam Category | | 2011 | 2012 | 2014 | 2015 |
|---|---|---|---|---|---|
| Similar Semantic | Template-based | 63.0% | 68.3% | 78.1% | 76.4% |
| | Paraphrase | 14.7% | 12.9% | 3.1% | 4.1% |
| Dissimilar Semantic | No-content | 8.4% | 0.3% | 0.2% | 0.3% |
| | Others | 13.9% | 18.5% | 18.6% | 19.2% |

as a whole may not have a common substring. Table I shows five (intentionally retrofitted) sample spam tweets from a much larger yet typical template-based campaign. We substitute the embedded URLs with the symbol $URL$ for brevity. The tweets in this campaign comprise three components: a celebrity name, an eye-catching action, and a URL. Each component has one or more choices of textual content. The number of unique spam messages that this template can potentially generate, therefore, increases quickly with the number of components.

We formally model the *true* spam template as a macro sequence $(m_1, m_2, ..., m_k)$. We define two types of macros: *dictionary* macros and *noise* macros. At the time of spam generation, a *dictionary* macro picks the textual content from a pre-defined list of choices. It is possible for a dictionary macro to have only one choice. In this case, the macro reduces to an invariant substring that all generated messages will contain. In comparison, we abstract any macro that does not convey any semantic meaning, but purely increases the message diversity or increases the chance of exposing the spam to more users, as a *noise* macro. The concatenation of the instantiation of macros constitutes a spam message.

We assume that a template shall contain at least one *dictionary* macro, while it may or may not contain any *noise* macro. However, we do *not* assume the existence of any invariant substring. Written in human language, a spam message is not restricted to any particular expression to present a semantic meaning. We have also observed spam template without invariant substring in our data. This relaxed assumption is one of our major differences from the existing template generation work [10], [11] that relies on invariant substrings.

**Paraphrase Category** consists of spam tweets that share the same semantic meaning but cannot be uniformly divided into semantically equivalent segments. Meanwhile, the tweets do not share regular wording. We denote them as "paraphrase" spam. Table II shows three retrofitted sample tweets from a paraphrase spam campaign that promises URLs using topics of a popular singer. The wording is highly diverse. Even for the singer's name, spammers use a variety of terms. Also, the tweets may not be grammatically correct and complete sentences, which contrast from template-based spam.

**No-content Category** does not contain any semantically meaningful sentence. Tweets in this category contain only one URL, followed by a long list of popular keywords and hashtags. Obviously, the spammers rely on the keywords and hashtags to increase the chance to expose the URLs to users when they browse tweets by topics.

**Other Spam** consists of the remaining spam that we have not systematically categorized. However, we do capture several interesting spam types. For example, a significant amount of remaining spam appears to be news snippets. Thomas et al. also identified spammers that exclusively retweet from major news accounts [6] in their earlier Twitter study. Since their textual content is legitimate, the detection of such spam is beyond the capability of content-based approaches.

## C. Template-based Spam Keeps Dominating

We further categorize the spam generated in January, 2012 and October, 2014 in the same way. Table III provides the popularity of four spam categories in June/July, 2011, January, 2012, October, 2014, and January, 2015. Template-based spam remains to be the most popular category in 2012, 2014 and 2015, with its percentage increasing to 68.3%, 78.1%, and 76.4%, respectively. The no-content category almost vanishes. Its percentage dramatically drops to 0.3%, 0.2%, and 0.3%, respectively. It is possible that the no-content category exhibits strong patterns and can be easily blocked. The increasingly popular template-based spam indicates that our detection method with focus on spam template generation is effective to combat modern OSN spam.

## III. TANGRAM: TEMPLATE-BASED SPAM DETECTION SYSTEM

In this section, we present Tangram, an accurate and fast template-based spam detection system. We first formulate the notions of template, template matching and template generation. Next, we detail the online Tangram system.

## A. System Design Overview

Tangram builds template-based spam detection on top of existing detection methods toward higher accuracy and speed. It generates the underlying templates of spam detected by various existing methods. It then uses the templates to accurately, quickly match and detect spam. Figure 1 depicts the Tangram workflow. It takes a stream of raw messages as input, and classifies them as either spam or legitimate online. After the classification, spam is filtered, while legitimate messages pass through. Two components can classify messages: the template matching module and the auxiliary spam filter. The template matching module, along with the template generation technique, is our major contribution. The auxiliary spam filter, on the other hand, supplies training spam messages. It can be any deployed spam filter, e.g., a blacklist spam filter.

**Template Matching and Template Generation.** We define a *template* to be a sequence of macros of two types, dictionary and noise (Section II-B). We represent a dictionary macro as a set of values separated by "|" and a noise macro as ".*". Thus, templates produced by Tangram are naturally encoded as regular expressions, specifically concatenations of "|" clauses and ".*"s. *Template matching* matches a given message against the corresponding regular expression. A successful template match implies the tested message instantiates the template, and should be flagged as spam. We define *template generation* as the task of inferring the template's regular expression representation from a set of observed spam instances.

Initially the template matching module is not equipped with any template, so all messages will pass through. However, if a message is blocked by the auxiliary spam filter, it is

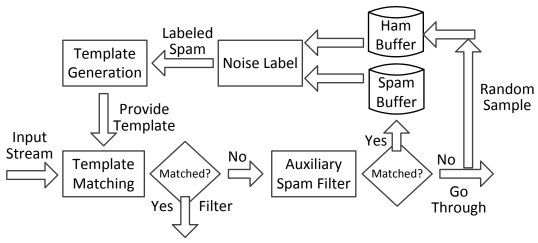


Fig. 1. Tangram framework: The template generation and matching overview.

treated as an instantiation of an unforeseen template, and is saved in the spam buffer. Once the number of messages in the spam buffer exceeds a predefined window size threshold $t$, the system invokes the template generation procedure, and deploys the newly generated templates in the template matching module. As spam categorization demonstrates, spam tweets are with or without underlying templates. We first preprocess spam tweets in the buffer into campaigns. Messages belonging to no campaign are not template-based or its same-template messages are not sufficient enough. We then feed only messages in campaigns to template generation. For the left spam tweets in the buffer, we wait for 10 times of window size until we evict them from the buffer. Whenever a newly generated template happens to be highly similar to an existing one, we will merge their regular expressions.

The template generation first identifies the subset of spam messages sharing the same template (Section III-C). These messages are tokenized into sequences of words. After executing noise detection (Section III-D), we are left with spam content generated by dictionary macros. We divide every message into the same number of segments. Each segment, containing zero or more tokens, corresponds to one macro in the template. We then construct the macro by combining the segment's unique strings across messages with "|". The concatenation of the macros for all segments constitutes the complete template.

Inferring the number of segments and which tokens belong to which segment are key challenges in template generation[1]. We use the heuristic of preferring more compact templates (i.e., shorter regular expressions) that match all of the input spam messages without using wild cards. This heuristic follows the traditional approach of preferring simpler descriptions to more complex ones; our experiments validate its effectiveness. Furthermore, finding the shortest template for a set of messages is an NP-hard problem (Appendix A in [1]). We develop a practical approximation as follows.

## B. Single Campaign Template Generation

For ease of presentation, we first introduce the approach to generate a single template given spam instantiating the *same* underlying template. It is the basis of Tangram. We use the template generation process of the campaign in Table I as a running example to elaborate our approach Tables IV-VII). We expand the approach to generate templates on a mixture of spam instantiating multiple templates in Section III-C.

[1]Common techniques for segmenting text into chunks from Natural Language Processing (NLP) cannot easily apply to our segmentation task because 1) it is difficult to use standard NLP tools on OSN text [20] and 2) our segments often do not correspond to typical NLP segments like noun phrases.

A strength of our approach is generating templates without any invariant substring. However, we do expect that some non-trivial *subset* of a campaign will share a common substring, because the dictionary macro may instantiate to the same textual content when multiple spam messages are generated. This property helps infer the correspondence of segments between messages. E.g., if we observe the first two messages in Table I, it strongly indicates that "Big Name A" and "Celebrity B" are two instantiations of the same macro. This indication holds for the campaign even if some messages do not have the substring "an eye-catching action -".

Since substrings shared by *subsets* of a campaign are crucial for template inference, a naive alternative is to break a campaign apart so that each part contains an invariant substring, then reuse the existing template generation algorithm. Unfortunately, such a naive alternative cannot capture as much spam as our technique does. Using the sample campaign in Table I for example, if we break the campaign into two parts, the first two messages and the last three messages, the templates generated by the naive alternative cannot capture the unobserved message starting with "RIP Celeb C", whereas our technique can detect such case.

We systematically exploit such substrings shared by subsets of a campaign in three steps, *common supersequence computation*, *column concatenation*, and *regular expression representation*.

**Common Supersequence Computation.** The first step is to compute the messages' common supersequence. Shortest common supersequence is an NP-hard problem (Appendix A in [1]). We use an approximation algorithm named Majority-Merge [21] because of its simplicity. It takes $n$ sequences as input and initializes the supersequence, $s$, as an empty string. It iteratively chooses the majority of the leftmost tokens of the input sequences, denoted as $a$, and appends $a$ to $s$. Meanwhile, the leftmost $a$ is deleted from the input sequences. It repeats this step until all sequences are empty and outputs $s$.

For ease of understanding the Majority-Merge algorithm, Table IV instantiates its beginning six execution steps over the five sample spam tweets in Table I. In step 1, both "Big" and "Celebrity" represent the majority of the leftmost tokens of the input tweets. We, however, assign $a$ with "Big" because its corresponding input tweet precedes the one that "Celebrity" corresponds to. (This rule applies also to subsequent steps.) We first delete "Big" from tweets prefixed with it (i.e., the first one and the third one). We then use the updated tweet set as the input of step 2. Again, step 2 finds two tokens, "Name" and "Celebrity", that dominate the majority of the leftmost tokens. We select "Name" as $a$ according to the rule in step 1. After six steps of similar operation, the (intermediate) output supersequence $s$ becomes "Big Name A Celebrity B an".

In the final output supersequence, each token is trivially a substring shared by some subsets of the campaign. Desirable substrings are $i)$ shared by large subsets and $ii)$ long. We achieve goal $i)$ by producing a shorter supersequence via the following two phases: matrix representation and matrix column reduction.

*Matrix Representation.* We build a matrix during the execution of Majority-Merge algorithm. Table V shows such a table

TABLE IV
THE BEGINNING SIX EXECUTION STEPS OF THE MAJORITY-MERGE ALGORITHM WITH THE FIVE SAMPLE SPAM TWEETS IN TABLE I AS INPUT SEQUENCES. SUPERSEQUENCE $s$ IS INITIALIZED AS AN EMPTY STRING. TOKEN $a$ CORRESPONDS TO THE MAJORITY OF THE LEFTMOST TOKENS OF INPUT SEQUENCES.

| Step 1: $a$ = Big, $s\|\|a$ = Big | Step 2: $a$ = Name, $s\|\|a$ = Big Name | Step 3: $a$ = A, $s\|\|a$ = Big Name A |
|---|---|---|
| **Big** Name A an eye-catching action - $URL$ | **Name** A an eye-catching action - $URL$ | **A** an eye-catching action - $URL$ |
| Celebrity B an eye-catching action - $URL$ | Celebrity B an eye-catching action - $URL$ | Celebrity B an eye-catching action - $URL$ |
| **Big** Name A offensive content , look at this video $URL$ | **Name** A offensive content , look at this video $URL$ | **A** offensive content , look at this video $URL$ |
| Celebrity B offensive content , look at this video $URL$ | Celebrity B offensive content , look at this video $URL$ | Celebrity B offensive content , look at this video $URL$ |
| RIP Celeb C offensive content , look at this video $URL$ | RIP Celeb C offensive content , look at this video $URL$ | RIP Celeb C offensive content , look at this video $URL$ |
| Step 4: $a$ = Celebrity, $s\|\|a$ = Big Name A Celebrity | Step 5: $a$ = B, $s\|\|a$ = Big Name A Celebrity B | Step 6: $a$ = an, $s\|\|a$ = Big Name A Celebrity B an |
| an eye-catching action - $URL$ | an eye-catching action - $URL$ | **an** eye-catching action - $URL$ |
| **Celebrity** B an eye-catching action - $URL$ | **B** an eye-catching action - $URL$ | **an** eye-catching action - $URL$ |
| offensive content , look at this video $URL$ | offensive content , look at this video $URL$ | offensive content , look at this video $URL$ |
| **Celebrity** B offensive content , look at this video $URL$ | **B** offensive content , look at this video $URL$ | offensive content , look at this video $URL$ |
| RIP Celeb C offensive content , look at this video $URL$ | RIP Celeb C offensive content , look at this video $URL$ | RIP Celeb C offensive content , look at this video $URL$ |

TABLE V
THE INITIAL MATRIX BUILT BY THE COMMON SUPERSEQUENCE COMPUTATION PROCESS. THE HEADER ROW IS THE COMPUTED SUPERSEQUENCE. EACH REMAINING ROW CORRESPONDS TO ONE INPUT SENTENCE.

| Big | Name | A | Celebrity | B | an | eye-catching | action | - | $URL$ | offensive | content | , | look | at | this | video | $URL$ | RIP | Celeb | C | offensive | content | , | look | at | this | video | $URL$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Big | Name | A | $\varepsilon$ | $\varepsilon$ | an | eye-catching | action | - | $URL$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ |
| $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | Celebrity | B | an | eye-catching | action | - | $URL$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ |
| Big | Name | A | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | offensive | content | , | look | at | this | video | $URL$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ |
| $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | Celebrity | B | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | offensive | content | , | look | at | this | video | $URL$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ |
| $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | RIP | Celeb | C | offensive | content | , | look | at | this | video | $URL$ |

TABLE VI
THE INTERMEDIATE MATRIX AFTER THE MATRIX COLUMN REDUCTION STEP. THE HEADER ROW IS A SHORTER SUPERSEQUENCE.

| Big | Name | A | Celebrity | B | an | eye-catching | action | - | RIP | Celeb | C | offensive | content | , | look | at | this | video | $URL$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Big | Name | A | $\varepsilon$ | $\varepsilon$ | an | eye-catching | action | - | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $URL$ |
| $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | Celebrity | B | an | eye-catching | action | - | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $URL$ |
| Big | Name | A | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | offensive | content | , | look | at | this | video | $URL$ |
| $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | Celebrity | B | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | offensive | content | , | look | at | this | video | $URL$ |
| $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | RIP | Celeb | C | offensive | content | , | look | at | this | video | $URL$ |

TABLE VII
THE FINAL MATRIX AFTER COLUMN CONCATENATION. THE HEADER ROW REPRESENTS THE REGULAR EXPRESSION FOR THE UNDERLYING TEMPLATE.

| (Big Name A \| Celebrity B \| RIP Cele C) | (offensive content , look at this video \| an eye-catching action - ) | $URL$ |
|---|---|---|
| Big Name A | an eye-catching action - | $URL$ |
| Celebrity B | an eye-catching action - | $URL$ |
| Big Name A | offensive content , look at this video | $URL$ |
| Celebrity B | offensive content , look at this video | $URL$ |
| RIP Cele C | offensive content , look at this video | $URL$ |

for the example campaign in Table I. The header row is the final supersequence output by the Majority-Merge algorithm. Each of the remaining rows represents one input sequence, that is, one spam tweet in Table I. If the $i$th sequence is picked in step $j$ for extracting token $a$, the cell at row $i$, column $j$ will be assigned the token $a$. (We also call the token $a$ as column label.) Otherwise, the cell will be assigned an empty string, $\varepsilon$. Take the first column in Table V for example. It corresponds to the first step of the Majority-Merge algorithm (i.e., step 1 in Table IV). Since the chosen token $a$ is "Big", which leads the first and the third tweets, the first row and the third row are labeled with "Big"; other rows are labeled with "$\varepsilon$". Naturally, the concatenation of labels of each row is exactly the row's corresponding input sequence. We denote this property as the *supersequence property*.

*Matrix Column Reduction.* To produce a shorter supersequence, we need to merge columns that share the same label, while maintaining the supersequence property. After merging two cells from two columns, the new cell will be assigned the column label if either cell before merging has been assigned so. Take the two columns sharing "offensive" in Table V for example. After merging them, the first two new cells will be assigned $\varepsilon$ because all corresponding cells are $\varepsilon$. On the other hand, each of the other three new cells will be assigned "offensive", which is the column label of one corresponding cell before merging. These two columns are merged into one column containing "offensive" in Table VI. Without loss of generality, we state the three sufficient conditions that determine whether column $k$ can be merged into column $j$ without affecting the supersequence property (Appendix B in [1]). Note that the merging is directional, after which column $j$ is kept while column $k$ is deleted. $Condition\ i)$ column $j$ and column $k$ have identical label; $Condition\ ii)$ in any row at least one column is $\varepsilon$; and $Condition\ iii)$ if the cell at row $i$, column $k$ is not $\varepsilon$, all cells in row $i$, between column $j$ and column $k$ must be $\varepsilon$. Table VI shows the column merging result. Noticeably, the repeated columns of "offensive content, look at this video" is gone after the merging, yielding a more compact matrix representation.

**Column Concatenation.** To achieve goal *ii)* for obtaining *long* substrings shared by subsets of campaigns, we further concatenate the matrix columns obtained from the previous step. Column concatenation also operates on a column pair, after which each cell becomes the concatenation of the two corresponding cells. Different from column merging, column concatenation does not require the target columns to share identical label. It only requires that the value of the corresponding, non-$\varepsilon$ cells in the two columns has 1-1 mapping. For example, the first two columns in Table VI are concatenated because "Big" always maps to "Name", but the fifth and the sixth columns in Table VI cannot be concatenated because "B" maps to two values, "an" and $\varepsilon$.

The effect of column concatenation is two-fold. First, it moves multiple tokens into one cell, revealing the true template by assembling tokens (words) into word phrases. For example, the three separate columns "Big", "Name", and "A" in Table VI become one celebrity name in Table VII. Second, the cells on the same column after column concatenation may have different contents, like the first two columns in Table VII. This maps to the dictionary macro case, where different cell contents are different instantiation of the dictionary macro.

**Regular Expression Representation** converts the matrix into a regular expression to represent the generated template. We initialize the regular expression representation to be an empty string, $s$. Then we iterate through each column. If all the cells in the column share an identical value, we append the value to $s$. Otherwise, we make a "|" clause by concatenating all the unique values with "|", and append the clause to $s$. The header row of Table VII gives an example of the generated regular expression representation. Finally, we add a "^" and a "$" to the head and the tail of $s$ to respectively mark the beginning and the ending of a message.

## C. Multi-campaign Template Generation

We now expand single campaign template generation to multi-campaign scenarios over spam instantiating *different* templates or even without underlying templates. We first separate the spam into distinct campaigns automatically and then individually invoke the single campaign template generation.

We first use single-linkage clustering to group messages that share at least $k$ consecutive identical tokens, $k$ being a system parameter. The goal is to put semantically similar messages in the same cluster, while separating semantically different messages into different clusters. The transitive closure of these links forms our initial clustering. This clustering does *not* require every message pair in the cluster to share an invariant substring. We use a small training set of collected spam tweets to choose the value of $k$ experimentally. The value of $k$ is not sensitive to the training set size. For example, we test with size 10,000, 5,000 and 2,000 and obtain consistent results. With a training set of size 10,000, a loose threshold (e.g., $k = 3$) results in a big cluster containing 42% of the spam, while spam messages in this cluster have different semantic meanings like Lady Gaga, Apple product and so on. A tight threshold (e.g., $k >= 5$) results in a large number of small clusters, where multiple clusters share the same semantic meaning. For example, 9 out of the 20 largest clusters in the experiment should be merged. In comparison, $k = 4$ produces the best result in our experiments. We suggest that $k$ be empirically adjusted. Given a test dataset, one can first extract a subset of test tweets and run template generation over it with an initial $k$. If it results a large number of smaller clusters, we need to shrink $k$. Otherwise, we increase $k$ instead. When it goes to practical filtering of real-time tweet stream, we again track a subset of messages and assign an initial $k$ first. We then post-analyze the detection result of the subset. If many obvious spam tweets are not detected, we need to improve template generation precision by increasing $k$.

We then refine the clusters using the single campaign template generation algorithm. Intuitively, spam tweets from different campaigns will result in non-compact templates, a fact we utilize to identify which spam should be removed from a given cluster. We explain this process using the dataset in Table VIII as a running example. Specifically, we find a row to remove if the number of $\varepsilon$ is larger than a predefined threshold $w \times p$, where $w$ denotes the word count (except notations and URLs) of the dataset/matrix and $p$ is a systematical parameter. We set $p$ no larger than the reciprocal of average word count per column. The reason is as follows: after column concatenation (from Table VI to Table VII), we can treat words in each cell as one new larger word. Then $w$ times the reciprocal of average word count per column approximates the number of such new larger words. If $\varepsilon$ is more than such approximation, we consider the matrix as non-compact. Take Table VIII for example. It contains nine $\varepsilon$'s, which is larger than $43 \times 0.2 = 8.6$ and we need to remove certain rows and all-$\varepsilon$ columns to make it more compact. We first find the column with the most $\varepsilon$'s, that is, the fourth column. We then remove any row corresponding to non-$\varepsilon$ words in the fourth column, that is, the last row. After deletion, the fourth column contains only $\varepsilon$, which should be removed as well. We repeat the above process over the new matrix and get Table VII.

## D. Noise Labeling

Spam tweets often mention other users, popular terms and hashtags unrelated to the semantics of the rest of the tweet [22]. Such content helps expose spam to a larger audience, because users may search or browse tweets by topic. It also diversifies spam and makes detection difficult. We refer to this type of content as *noise*. Popular forms of noise include celebrity names, TV shows, trending hashtags and many others. We next elaborate how noise affects template generation and design a model to automatically label noise given a small amount of easily, manually labeled noise as trained data. Once trained well, the model can accurately label noise tokens in real-time stream of spam tweets during Tangram execution.

Noise creates extra difficulties for template generation. If the generated template contains a segment of noise, the template will be too "specific", in the sense that it cannot match the spam with a different sequence of noise terms. In addition, spam instantiating different templates may coincidentally share an identical sequence of noise terms. It increases the chance to mislead the template generation module so that it attempts to extract a single template for them. Thus, we first perform a pre-processing step to identify noise tokens in the tweet, and then effectively ignore them when generating the template (i.e., we replace them with .*, a wildcard that matches anything).

We treat noise detection as a sequence labeling task, in which the goal is to automatically label each token in the tweet as noise or non-noise. We employ a standard sequence-labeling

TABLE VIII
THE EXAMPLE MATRIX WHICH IS NOT SOLID AFTER COLUMN CONCATENATION.

| (Big Name A \| Celebrity B \| RIP Cele C) | (offensive content , \| an eye-catching action - ) | look at this video | error message | $URL$ |
|---|---|---|---|---|
| Big Name A | an eye-catching action - | $\varepsilon$ | $\varepsilon$ | $URL$ |
| Celebrity B | an eye-catching action - | $\varepsilon$ | $\varepsilon$ | $URL$ |
| Big Name A | offensive content , | look at this video | $\varepsilon$ | $URL$ |
| Celebrity B | offensive content , | look at this video | $\varepsilon$ | $URL$ |
| RIP Cele C | offensive content , | look at this video | $\varepsilon$ | $URL$ |
| $\varepsilon$ | $\varepsilon$ | look at this video | error message | $URL$ |

approach, Conditional Random Fields (CRFs) [23]. The CRF is a model, learned from training data, that infers a label for each token in a given tweet. The model exploits regularities in the features of noise and non-noise tokens (detailed below), as well as regularities in label sequences.

The CRF requires identifying a set of features for each token that are relevant to the task. In our case, we found a set of features that appear to be highly indicative of noise. The key observation is that noise terms are popular, yet unrelated to each other and to other elements of the tweet. We would expect regions of noise to contain individual tokens that are common on Twitter, but transitions between tokens that are relatively uncommon. We capture these intuitions with three numeric features. Let $freq(s)$ represent the frequency of a string $s$, which we measure of a large set of unlabeled tweets. For each token $t_i$ in a tweet, we create the following three features in the CRF: $freq(t_i)$, $freq(t_i t_{i+1})^2/(freq(t_i)freq(t_{i+1}))$, and $freq(t_{i-1}t_i)^2/(freq(t_{i-1})\ freq(t_i))$. The first feature captures the popularity of the token $t_i$, whereas the second and third estimate how likely $t_i$ is to occur given the surrounding tokens. We processed these features into five discrete quantiles for incorporation into the CRF.

We further add four orthographic features to capture common elements of noise terms. They indicate whether $t_i$ is capitalized, is numeric, is a hashtag, or is a user mention @.

To train our CRF, we hand-labeled 1,000 tweets as training data, manually identifying each token as noise or non-noise. We then employed this learned model on each tweet before template generation. In a separate experiment on the labeled tweets, we found that our trained CRF correctly labeled an average of 92% of test-set tokens as noise or non-noise.

**Detection based on noise labeling.** Besides pre-processing tweets to facilitate template generation, noise labeling can also directly detect spam from another angle. Intuitively, we expect legitimate messages to have very few semantically unrelated noise terms, whereas spam contains much larger number of noise terms. We design a straightforward idea to use the percentage of noise terms in the message to distinguish spam. Our system classifies a tweet as spam if its percentage of noise terms is larger than a threshold $t$. In the experiments we set $t$ to be 75%. This threshold is relatively high and conservative, because we want to minimize the false alarm on legitimate tweets. However, as later results will show (Section IV-A), noise labeling favors only no-content spam, which accounts for less than tenth of spam dataset. We thus cannot rely on noise labeling for accurate spam detection.

## IV. Experiments

We evaluate Tangram using the labeled dataset in Section II-A as ground truth. The two major metrics that we use to evaluate the system are accuracy and speed. For accuracy, we primarily evaluate the following two aspects: $i)$ true positive rate, the ratio of correctly classified spam to the total number of spam, (We only count spam caught by template matching or noise detection as *true positives*.) and $ii)$ false positive rate, the ratio of legitimate but incorrectly classified messages to the total number of legitimate messages. (We count mis-detected legitimate messages as *false positives*.) Besides, we may occasionally report *false negatives*, which count spam missed by the two modules but labeled in the ground truth. Defining the ratio of such spam to the total number of spam in ground truth as false negative rate, we have that false negative rate + true positive rate = 1. For speed, we evaluate the template generation and matching latency. We feed the system with the collected tweets obeying their timestamp order to reflect the performance in real-world scenario. We conduct all experiments on a server with an eight-core Xeon E5520 2.2GHz CPU and 16GB memory.

Tangram needs an auxiliary spam filtering module to provide the initial set of spam messages to construct the underlying template. When a message reaches the auxiliary spam filter, it needs to be classified as either spam or legitimate with very little latency. We leverage an existing online OSN spam filtering tool [3] to provide training spam samples toward conducting a realistic evaluation. We reuse the same parameters reported in the paper. The auxiliary spam filter is *not* an oracle. It may mistakenly report legitimate messages as spam, or miss to report spam messages. Note that in what follows we evaluate the detection accuracy of only the modules proposed in this paper, but not the accuracy of the auxiliary spam filter. We will factor the effect of auxiliary filter's accuracy to Tangram in Section IV-D.

### A. Detection Accuracy

We test Tangram with spam window size $t$ = 1000, which means when the number of spam messages that slip through the template matching module but are blocked by the auxiliary spam filter reaches 1000, the system will invoke the template generation module to infer the underlying templates of the messages. The value of parameter $k$ is 4.

The results show that the TP rate for the most prevalent template-based spam achieves 95.7%. The system can also detect some non–template-based spam messages, because the system treats all messages as if they were template-based, and makes best-effort detection. As expected, the TP rate of such messages is lower than that of template-based messages. The overall TP and FP rate are 76.2% and 0.12%, respectively.

**True Positive Analysis.** Table IX reports a detailed breakdown of true positive rate into different spam categories. Tangram has two detection modules. Both modules perform

TABLE IX
THE DETECTION ACCURACY OF TANGRAM AND TWO EXISTING SYSTEMS COMPARED IN SECTION IV-B. THE LAST COLUMN SHOWS THE ACCURACY IF WE COMBINE TANGRAM AND THE SYNTACTICAL APPROACH.

| **System** | Template Based | | Syntactical | Tangram + |
|---|---|---|---|---|
| | Tangram | Judo | Clustering | Syntactical |
| **Spam Category** | | | | |
| Template-based | 95.7% | 32.3% | 70.1% | 98.4% |
| Paraphrase | 51.0% | 52.2% | 51.4% | 70.1% |
| No-content | 73.8% | 41.9% | 67.0% | 83.1% |
| Other | 18.4% | 30.4% | 43.2% | 44.7% |
| Overall TP | 76.2% | 35.9% | 63.3% | 85.4% |
| FP | 0.12% | 5.0% | 0.27% | 0.33% |

well on the specific spam category that they are designed for. The template generation/matching module can detect template-based spam with 95.7% TP rate (336,849 out of $558,706 \times 63\%$). The noise detection module can detect no-content spam with 73.8% TP rate (34,635 out of $558,706 \times 8.4\%$). Unfortunately, the true positive rate of the other two spam categories is lower. About 80% of the false negatives (spam misclassified as legitimate with a rate of 1 - TP rate) belong to the other two categories.

**False Positive Analysis.** Since the labeling approach we use to build the ground truth may miss to label true spam tweets (Section II-A), We further compare the true positives against the detected tweets that are not labeled. We observe that spammers frequently attach *Retweet* marks (RT @$username$) and *Mentions* (@$username$) at the beginning of tweets, as well as noise words after the embedded URL. Hence, we remove all the noise and acquire the stem of spam tweets. Any tweet that shares the same stem with spam tweets is also regarded as spam. The comparison reveals that 15,271 (0.12%) tweets reported by Tangram are neither labeled as spam, nor sharing the same stem with spam tweets. They represent the false positives that our system incurs. We thus use it as a post-processing step in the evaluation, rather than adopting it in the system design.

Among the false positive tweets, 42.0% of them are caused by overly general spam templates. Another 21.7% of them are popular tweets like birthday wishes for Nelson Mandela. These popular tweets are mistakenly reported as spam by the auxiliary filter, so templates are generated to match them.

**Template Analysis.** To understand how spammers develop spam templates empirically in a realistic dataset, Table X shows the size, in terms of percentage, and the underlying templates, represented as regular expressions, of the five largest campaigns in the true positives. The actual URLs are replaced by the "{URL}" symbol. An $\varepsilon$ means the dictionary macro may instantiated as an empty string. An "..." means that the dictionary macro has more options that are not shown due to readability and space consideration.

*Size.* We do not observe dominant templates in Twitter. The most popular template matches 11.1% of true positives. The top ten most popular templates match 50.5% of true positives.

*Use of Noise.* Although in theory spammers can insert noise at any position in the template, practical spam messages have noise at either the beginning or the end, or both. Noise at the beginning is usually *Mentions* (@$username$) and *Retweet* marks (RT @$username$). There can be multiple consecutive Mentions and Retweet marks at the message beginning. Noise at the end contains primarily *Hashtags* (#XXX, XXX being a topic) and popular terms like celebrity names and TV shows.

*Message Diversity.* As the number of dictionary macro increases, the number of unique messages the template can generate increases quickly. All templates in Table X adopt multiple dictionary macros.

## B. Detection Accuracy Comparison with Existing Work

We limit the direct experimental comparison to only the approaches that examine the message content to detect spam.

**Syntactical clustering + machine learning.** We first compare with a recent spam detection work that adopts syntactical message clustering and supervised machine learning [3] (denoted as the syntactical clustering approach hereafter) in detail. The two systems share similar design goals. In addition, the existing approach is used as the auxiliary spam filter in our experiment. Hence, it is crucial to quantify the detection accuracy gains over directly using the existing system. We run the system using the same dataset on which we test Tangram.

The syntactical clustering approach achieves an overall detection accuracy of 63.3% TP rate and 0.27% FP rate. The true positive rate obtained by the syntactical clustering approach on our data is lower than the reported number in [3]. The reason is that our spam labeling approach labels more spam tweets as ground truth, which the syntactical clustering approach does not detect. In contrast, Tangram achieves a substantial improvement on both the TP rate (to 76.2%) and the FP rate (to 0.12%). Table IX lists the detailed accuracy comparison. The difference between the spam detected by these two systems indicates that they can potentially complement each other. This simple integration suffers from increased FP rate of 0.33%, but can boost TP rate to 85.4%.

**Judo.** To validate that our template generation technique is more tailored to OSN spam detection, we also compare our work with a recent email spam detection system called *Judo* [10]. *Judo* detects email spam based on template generation. We have already presented the difference between the two systems analytically by elaborating the difference in the critical system assumptions, i.e., invariant substring in template and quality of training samples. We further demonstrate their difference in experimental results, shown in Table IX, column "Judo". Different from our system, *Judo* requires training set that contains *pure spam* generated by the *same* underlying template. We implement the template generation mechanism of *Judo* as described in [10], and test the detection accuracy using the same dataset. Even with small window size (10 spam messages), the generated templates can only achieve 35.9% TP rate. The TP rate further drops to 10.6% if the window size is increased to 20. On the other hand, the FP rate is high (5.0%). It shows that real-world OSN trace breaks the crucial assumptions of *Judo*. As a result, *Judo* achieves extremely high accuracy in email spam detection, but does not perform well for OSN spam detection.

## C. Detection Accuracy Comparison with URL Blacklists

It is well known that most spam fools people via deceitful (plain or shortened) URLs [5]. Such URLs usually direct to

TABLE X
THE TEMPLATES, REPRESENTED AS REGULAR EXPRESSIONS, OF THE FIVE LARGEST CAMPAIGNS.

| **Size (%)** | **Spam Template** |
|---|---|
| 11.1% | .* (I wager\|My my,) you (cannot\|$\varepsilon$) (defeat\|$\varepsilon$) this . {URL} .* |
| 7.2% | .* The (people\|folks\|$\varepsilon$) at my (place\|location\|$\varepsilon$) are groveling for this ! {URL} .* |
| 6.4% | .* You (will not\|won't\|$\varepsilon$) (think\|believe\|$\varepsilon$) this. The (best\|greatest\|$\varepsilon$) (thing\|factor\|$\varepsilon$) (because\|since) slice bread. {URL} .* |
| 5.0% | .* (Cool\|Wow) , I (by no means\|in no way) (found\|noticed\|...)(people\|anyone\|...)(do that\|$\varepsilon$) (just before\|prior to) . {URL} .* |
| 4.1% | You (will not\|won't\|$\varepsilon$) (think\|believe\|$\varepsilon$) the (issues\|points\|things) they do on this (site\|web page\|web-site\|...) . {URL} .* |

websites that advertise scams, phish one's credentials, or propagate other malicious contents. An intuitive countermeasure might detect spam using blacklisted URLs. If a tweet contains a blacklisted URL, it is highly likely that the tweet is not benign. We verify such intuition on the following popular websites that provide URL blacklist query service—Bitly [24], Virustotal [25], and Wepawet [26].

We collect a more recent dataset with 9 million tweets from Twitter in October, 2014. Among spam tweets therein, 99.2% end with a shortened URL. We redirect all the URLs in the spam tweets to see whether the landing page has been suspended. Table XI summarizes the accuracy comparison. The results show that the ability of Bitly is limited—only 50.0% of spam tweets can be detected via suspended URLs on Bitly. Then we perform similar URL checking against Virustotal and Wepawet. They respectively reveal 50.9% and 33.3% of the spam. Note that the detected spam tweets via these websites are highly redundant; the overall detection accuracy is 58.1%. On the other hand, Tangram can detect them with the TP rate of as high as 77.9%. We run Tangram again over the January 2015 dataset. It yields a consistently satisfactory TP as in October 2014 dataset, which is 71.2%.

## *D. Effect of Auxiliary Filter Quality*

Since the auxiliary spam filter essentially provides training samples for template generation, it is crucial to understand how the accuracy of the auxiliary spam filter affects Tangram's detection accuracy. As aforementioned, we focus on the accuracy of our proposed modules—template matching and noise detection. We first try to analytically capture how their accuracy varies with that of auxiliary filter. Let $T_\mathrm{s}$ and $T_\mathrm{l}$ represent the number of spam tweets and the number of legitimate tweets in the ground truth. Let $S_\mathrm{s}$ and $S_\mathrm{l}$ respectively denote the count of spam messages caught by template matching or noise detection and the count of legitimate tweets mis-classified as spam. Then $S_\mathrm{s} + S_\mathrm{l}$ denotes the count of all spam tweets detected by our modules of template matching and noise detection, for which we have $\mathrm{FP} = \frac{S_\mathrm{l}}{T_\mathrm{l}}$ and $\mathrm{FN} = 1 - \frac{S_\mathrm{s}}{T_\mathrm{s}}$. Since $T_\mathrm{l}$ and $T_\mathrm{s}$ are invariants from ground truth, we next analyze how $S_\mathrm{l}$ and $S_\mathrm{s}$ and therefore FP and FN are affected by auxiliary filter's $\mathrm{FP}_\mathrm{aux}$ and $\mathrm{TP}_\mathrm{aux}$. (Note that we have $\mathrm{FN}_\mathrm{aux} = 1 - \mathrm{TP}_\mathrm{aux}$.) Let $S_\mathrm{l} = S_\mathrm{l}^\mathrm{template} + S_\mathrm{l}^\mathrm{noise}$, where $S_\mathrm{l}^\mathrm{template}$ and $S_\mathrm{l}^\mathrm{nosie}$ denote the count of legitimate tweets mis-classified as spam by respectively template matching and noise detection. $S_\mathrm{l}^\mathrm{nosie}$ is insensitive to auxiliary filter as it needs no training samples. Only $S_\mathrm{l}^\mathrm{template}$ varies with auxiliary filter's $\mathrm{FP}_\mathrm{aux}$ and $\mathrm{TP}_\mathrm{aux}$. It further falls into two parts: one is legitimate messages in $T_\mathrm{l}$ matched by templates generated from $\mathrm{FP}_\mathrm{aux} T_\mathrm{l}$ messages; the other is legitimate messages in $T_\mathrm{l}$ matched by templates generated from $\mathrm{TP}_\mathrm{aux} T_\mathrm{s}$ messages. As previous results show, the second part should be small. It is thus $\mathrm{FP}_\mathrm{aux}$ that affects more on our modules' FP. The larger $\mathrm{FP}_\mathrm{aux}$ is, the larger FP tends to be. Since it is hard to deduce the exact number of legitimate messages in $T_\mathrm{l}$ matched by templates generated from $\mathrm{FP}_\mathrm{aux} T_\mathrm{l}$ messages, we choose to empirically evaluate the effect of $\mathrm{FP}_\mathrm{aux}$ shortly.

TABLE XI
THE DETECTION ACCURACY OF TANGRAM AND OF SPAM DETECTION BASED ON ONLINE URL BLACKLISTS. B+V+W = BITLY + VIRUSTOTAL + WEPAWET.

| Tool | Tangram | Bitly | Virustotal | Wepawet | B+V+W |
|---|---|---|---|---|---|
| Accuracy | 77.9% | 50.0% | 50.9% | 33.3% | 58.1% |

Let $S_\mathrm{s} = S_\mathrm{s}^\mathrm{template} + S_\mathrm{s}^\mathrm{noise}$, where $S_\mathrm{s}^\mathrm{template}$ and $S_\mathrm{s}^\mathrm{noise}$ denote the count of spam tweets caught by template matching and noise detection, respectively. $S_\mathrm{s}^\mathrm{noise}$ is insensitive to auxiliary filter as it also needs no training samples. $S_\mathrm{s}^\mathrm{template}$ accounts for spam in $T_\mathrm{s}$ that matched by templates generated from $\mathrm{TP}_\mathrm{aux} T_\mathrm{s}$ messages. However, our modules' TP (= 1 - FN) does not have to increase with $\mathrm{TP}_\mathrm{aux}$. Although $\mathrm{TP}_\mathrm{aux}$ increases with the count of spam tweets caught by auxiliary filter, more detected spam may not promise more templates. Template matching thus may not match and detect more spam than it does using generated templates. Again, we will empirically evaluate how $\mathrm{TP}_\mathrm{aux}$ affects our modules' accuracy.

First, we mimic low true-positive–rate auxiliary filter by sampling 50%, 20% and 10% of spam uniformly at random to feed template matching. The resulting true positive rate is 64.9%, 64.9% and 63.0%, respectively. Given the randomness of tweet stream, such sampling decreases the number of spam tweets but not their generating templates. Our methods' true positive rate does not drop that much. We set auxiliary filter to have 0.5%, 1% and 2% false positive rate. The resulting false positive rate is 1.2%, 3.53% and 3.3%, respectively.

Second, we choose VirusTotal as auxiliary filter to further evaluate how our methods' accuracy (especially true positive rate) varies. As in Table XI, Virustotal's true positive rate is 50.9%. Using it as auxiliary filter, our methods yield a true positive rate of 57.7%. This is even lower than what we obtain using the above auxiliary filter with true positive rate of 10%. The reason is Virustotal completely misses detecting spam tweets generated by certain templates. Besides, our methods' false positive rate is 7.6% while Virustotal's is 4.3%.

In summary, Tangram's true positive rate drops only marginally if the auxiliary filter has low true positive rate that makes spam tweets with certain templates completely undetected. It is more sensitive to false positives from the auxiliary filter. Hence, in practice Tangram needs an auxiliary filter with a low false positive rate. This is a reasonable requirement, since we can tune the auxiliary filter to be conservative in reporting spam.

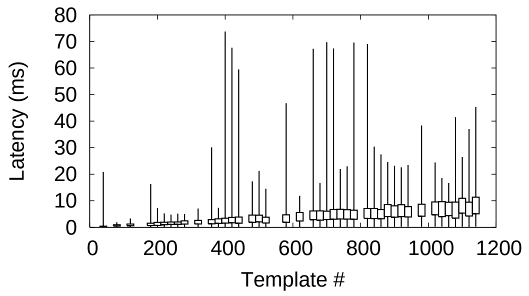


Fig. 2. The box plot of template matching latency as a function of the number of generated templates.

## E. Template Generation/Matching Speed

**Template matching.** The template matching latency incurred by Tangram is minimal and is not noticeable to users. Figure 2 plots the minimum, 25% quantile, 75% quantile and maximum of the template matching time as a function of the number of generated templates. We observe a large variance of template matching latency, because the time consumed for regular expression matching highly depends on the text being matched. Nevertheless, the largest latency in the entire dataset is less than 80ms. The overall trend is that the template matching latency, shown by the boxes representing the 25% quantile and the 75% quantile, grows slowly with the number of templates. Even with more than one thousand templates, the median template matching latency is only 8ms.

**Template generation.** It is crucial to throttle spam campaigns at their early stage. Hence, we measure how fast templates can be generated. The time to generate template depends on the number of buffered spam messages. In our experiment, the mean template generation time is only 2.3 seconds. Although slower than template matching, template generation is *not* the bottleneck of Tangram, since template generation is performed in parallel with template matching.

## F. Sensitivity for New Campaigns

We take the five largest campaigns, one of which matches the template instantiated by spam in Table I, and evaluate how fast Tangram reacts to newly emerged spam. We randomly select a small percentage of messages from each campaign, and use them as training samples to generate the template. We vary the percentage of training samples from 0.05% to 0.5%. The remaining messages serve as the testing set. We measure the true positive rate as the percentage of the testing set that the generated template can match. Figure 3 shows the results. We observe that all campaigns achieve almost 100% coverage even with only 0.15% of messages as training samples. Three campaigns have lower coverage when only 0.05% of messages are used to generate the template, because the system has not observed all possible values of dictionary macros due to insufficient training samples. Nonetheless, the coverage quickly climbs up to almost 100% when the percentage of training samples increases. The result indicates that when new spam campaigns emerge, the system can react quickly and generate effective templates to throttle them.

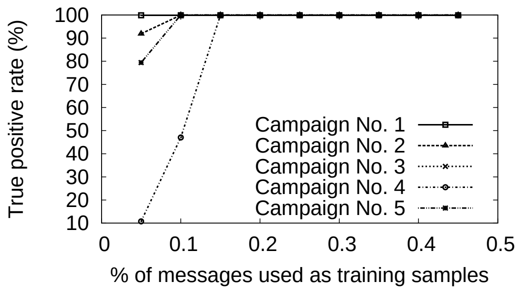


Fig. 3. The TP rate when template generation is performed separately for each campaign, varying the size of training set. Most observation points reach 100% TP rate.

## G. Accuracy on Facebook Data

To test Tangram's generality on other OSNs, we collect 4.7 million comments from public Facebook pages generated from January 2012 to April 2013, and run Tangram on the Facebook data. We use two well-known blacklists, Mcafee siteadvisor [27] and myWoT [28], to label the ground truth: any comments embedded with blacklisted URLs are labeled as spam. The dataset contains 6,337 spam comments embedded with blacklisted URLs. Tangram achieves 77.8% true positive rate and 0.08% false positive rate. In particular, the spam template matching module and the syntactic clustering module contribute 72.1% true positive rate and 64.6% true positive rate, respectively. This illustrates that our system yields promising results on other OSNs as well.

## V. Beyond Spam Detection: Inferring Spammer Strategy

While the infrastructure for email spam generation utilizing botnets is well-known to the research community [8], [9], [29], less is known about OSN spammers. One key aspect of OSN spammers is that they need to control a large number of accounts registered with the legitimate OSN service. To the best of our knowledge, there is no systematic study that reveals how spammers obtain and manage the accounts.

In this section, we present three questions regarding the OSN spammers' strategy and our inference methods to answer them. The findings can be used to design more features for detection of spam and spamming accounts, as well as providing situational-awareness of the attacks.

### A. Question 1: Are spamming accounts created in burst?

We seek the answer using the numeric ID that uniquely identifies every account, instead of the raw account creation time. The main reason is that more recent time period is more densely populated with account creation due to Twitter's exponential growth. As a result, any account created recently will seem to be created in burst comparing with accounts created in earlier years. It is difficult to adjust this artifact to make unbiased observation. As an alternative, IDs are assigned based on the account creation time, *i.e.* , accounts with smaller IDs are created earlier than those with larger IDs. We first validate this hypothesis in our collected dataset. We sort the accounts that appear in our dataset by their IDs and compare the creation time between each account and the account after it in the sorted list. Only 2,021 out of 4.2 million accounts are

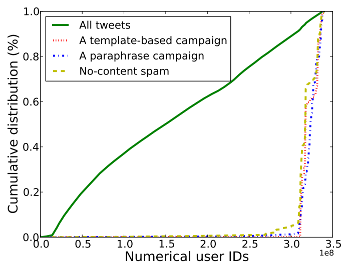


Fig. 4. The cumulative distribution of account IDs.

created later than the accounts after them, which is a negligible portion. A more rigorous Spearman correlation test shows a 0.999 correlation coefficient between the ID and account creation time. Hence, we can use the order of account IDs to represent the order of account creation time. Naturally, we can confirm that spamming accounts are created in burst if we observe spamming account IDs concentrating in some ID ranges, and reject it otherwise.

**Finding 1: Spamming accounts are created in burst.** We plot the cumulative distribution of all account IDs in Figure 4. The straight diagonal line represents a uniform distribution. We also plot the cumulative distribution of account IDs that generates the template-based campaign, the paraphrase campaign and no-content spam, respectively. Most of the spamming accounts are new accounts with large IDs between 311082373 and 336999761. The curves of the template-based campaign and the no-content campaign contain two steep segments and a flat segment in between. The steep segments are between ID range (311082373, 319434660) and (329186431, 336999761). They reflect the two periods when spamming accounts are created in burst.

A potential application of the bursty nature of spamming account creation is to identify ID ranges heavily populated with spamming accounts. For example, the template-based campaign shown in Figure 4 contains 30,048 unique spamming accounts. However, the campaign identifies the ID range containing 229,457 unique accounts, with 84,499 (36.8%) spamming accounts. The range is a strong indicator for s-pamming account classification.

## *B. Question 2: Which campaigns are launched by the same organization?*

Since the URLs embedded in spam reflect spammers' goal, *i.e.* , the product, scam or fraud being promoted, the number of shared URLs between campaigns shows how closely different campaigns are related to each other. Two campaigns with highly overlapping URL sets should be launched by the same spammer. To systematically study the connection among campaigns, we extract the set of embedded URLs of each campaign, and define the similarity between two campaigns as the *Jaccard similarity coefficient*, *i.e.* , the size of URL set intersection divided by the size of URL set union. A similarity of 0 means the two campaigns share no common URL. A similarity of 1 means the two campaigns have exactly the same URL set. With this definition, we cluster the detected campaigns. If we can find clusters of campaigns, it serves as strong evidence that they are launched by the same organization.

**Finding 2: It is likely that one organization launches multiple campaigns that contain the majority of spam tweets.** We conservatively set the similarity threshold as 0.8, requiring that the similarity of *all* campaign pairs in a cluster is greater than 0.8. We obtain a large cluster of 18 campaigns using this high threshold. This cluster contains 63.2% of spam tweets and 57.8% of spamming accounts in our dataset.

Different campaigns in this cluster promoting a highly overlapping set of URLs indicates that they are orchestrated by the same organization. Additional evidence is that the account IDs for all 18 campaigns in this cluster share highly similar distribution. The distribution of one of them is shown by the red dotted curve in Figure 4. We omit the curves for other 17 campaigns for the interest of space. It means that the spamming accounts used to generate these campaigns are likely to be registered in the same two batches.

## *C. Question 3: How does an attacker select OSN spamming accounts to launch campaigns?*

If a spammer launches multiple campaigns while controlling a pool of available accounts, he can either select the accounts in a coordinated way to launch different campaigns, or launch campaigns independently from each other. The extreme case is to select accounts uniformly randomly from the pool. Answers to this question involve independence test and uniformity test.

The standard test to check whether event $A$ and $B$ are independent is to test whether $prob(A \cdot B)$ equals to $prob(A) \times prob(B)$. In our context, event $A$ is that a spamming account is used to launch a campaign. Event $B$ is that the account is used to launch another campaign. Event $A \cdot B$ is that the account launches both campaigns. In order to compute the probabilities, we also need the pool of available accounts. If we find multiple campaigns controlled by the same organization from *Question 2*, we can use the union of accounts involved with the campaigns to represent the available account pool. The dynamic nature of the pool due to spamming account attrition adds extra complexity. We divide the campaigns into daily chunks, and compute the probabilities daily. We make an implicit assumption that the account attrition within a day does not significantly impact the probability computation. For each campaign pair, we compute expected and observed number of overlapping accounts, and employ standard $\chi^2$ test with a significance level of 0.5% to test the independence.

To determine whether spammers select accounts uniformly randomly from the pool to launch campaigns, we cast a distribution checking problem. For each campaign, we extract the set of IDs that launch it, and test whether the IDs are uniformly distributed in the range of all available IDs. $\chi^2$ test is well-suited for this task as well. Accordingly, we need to group available IDs into bins first. Next, we can compute the expected and observed number of IDs selected from each bin, and complete the $\chi^2$ test. $\chi^2$ test requires the expected value $E_i$ for any bin to exceed 5. Given the large number of spamming accounts, we put every 100 consecutive account IDs into one bin. In this way, we satisfy the expected value constraint and to have sufficient number of bins simultaneously.

**Finding 3: Accounts launching different campaigns are**

**chosen independently from the pool. However, they are *not* uniformly randomly selected.** We investigate the single active organization that controls 57.8% of spamming accounts and orchestrates 18 campaigns in particular. We obtain the daily chunks all campaigns, extract all involved accounts, and conduct $\chi^2$ tests on every campaign pair to check whether the account sets are independent with a significance level of 0.5%. Among 3344 tests that we conduct, 3058 (91.4%) tests pass, whereas 286 (8.6%) tests fail, meaning that we cannot reject the null hypothesis of independence in 91.4% of the test cases, with 99.5% conficence. In all the failed cases, the observed number of overlapping accounts is larger than the computed expected number. We hypothesize the reason to be fast spamming account attrition. Although we divide the campaigns into daily chunks, as reported in [6], 77% of spamming accounts are suspended in *less* than one day. As a result, the spammer needs to add new accounts into the available pool, which increases the pool size and decreases the expected number of overlapping accounts.

In addition, all $\chi^2$ tests on uniformity fail, indicating that the accounts are *not* uniformly randomly selected from the available account pool. We also visually confirm that the distribution does not appear uniform. The uniformity test failure shows that we cannot simply model spammers as selecting accounts from a static pool. Instead, an accurate model of spammers' account selection process must consider old account attrition and new account creation.

## VI. Discussions

In this section, we discuss how an attacker aware of Tangram design principle tries to evade spam detection or exploit it to, for example, block legitimate tweets. For evasion, an attacker could *i*) evade the detection of auxiliary filter, *ii*) send spam with different patterns from those currently in use, *iii*) add more paraphrase spam messages, *iv*) re-use legitimate content with malicious URLs, *v*) reduce the syntactic similarity among messages, or *vi*) produce non-textual spam. For exploitation, an attacker could issue a large volume of messages casted with templates generated from legitimate tweets he wants to block. Having elaborated on evasion concerns *i-iv* in the earlier version [1], we next investigate only evasion concerns *v-vi* and the exploitation concern.

**How to mitigate deliberate spam variety against template generation?** The possible way is to interfere with the clustering process by reducing the syntactic similarity among messages in one campaign. For example, spammers deliberately insert random meaningless words from a collection of popular words. Nonetheless, Tangram can defeat these attempts by extracting real-time popular words and conducting noise labeling before the clustering process.

**How to mitigate non-textual spam?** Since template generation works solely on textual messages, a sophisticated attacker might adopt non-textual spam to evade detection. For example, image-based spam has already emerged on Facebook [30]. To detect such spam, Jin *et al* proposes computing image similarity using their content features such as color histogram and color correlogram [30]. This is definitely beyond the scope of the paper; we suggest adopting non-textual spam detectors like the above one [30] as part of auxiliary filter.

**Can attackers exploit Tangram to filter legal tweets?** For example, the attacker can exploit the template generation process of Tangram. After extracting the template of tweets he wants to block, the attacker uses the template to more quickly generate similar tweets. When the volume of these tweets exceeds a certain threshold, the auxiliary spam filter detects the attacker as a spammer and regards his tweets as spam [2]. Based on current Tangram design, we take the detected spam for extracting template, which later helps block matching tweets although they might come from honest users.

We suggest that the above exploitation can be combated through a slight modification when deciding whether to use detected spam tweets for template generation. Specifically, the auxiliary spam filter double checks detected spam against recorded tweets to verify their content. If it conforms more to benign tweets, the auxiliary spam filter does not use it for template generation. This way, Tangram combats the exploitation for blocking benign tweets. We now analyze the feasibility of the countermeasure. In reality, it is normal for OSN providers to cache tweets. A straightforward yet inefficient method is then directly comparing detected spam tweets against recorded benign tweets. However, we prefer leveraging the clustering method Section III-C to classify benign tweets into clusters. The method tokenizes each tweet and then classifies tweets with $k$ consecutive identical tokens into the same cluster. The tweet pattern of a cluster we use for comparing with detected spam tweets by auxiliary filter is just the identical-token sequence of the cluster. This way, we achieve efficient verification of these spam tweets. Furthermore, when an attacker exploits Tangram to block benign tweets, such tweets likely emerged on the OSN for a while and were already cached. Thus, even if a tweet is from a spammer that generates an over-threshold volume of tweets, we still can double check the tweet's content against that of the cached ones. If it actually conforms to benign tweets, we can simply suspend the spammer yet without using the corresponding tweets for template generation.

We find that the above countermeasure does not affect spam detection accuracy. For spam tweets detected by auxiliary filter and matching with clustered benign tweets, we directly label them as spam without using them for template generation. Such spam tweets thus do not lead to mis-detection of benign tweets as spam and therefore incurs no false positive. Furthermore, we use an experiment to demonstrate that the above countermeasure leads to no false negatives. In particular, we randomly select 10,000 (5,000, and 2,000) tweets from the 2015 dataset. We first feed them to Tangram and cluster output benign tweets. We then from top five largest clusters randomly select a number of tweets with URLs; we further retrofit them by replacing their URLs with malicious ones and thus make 100 spam tweets. We then inject these retrofitted tweets in addition to 100 more conventional detected spam tweets into the dataset. Feeding the new dataset to Tangram, both types of spam tweets are successfully detected. All the 100 retrofitted tweets are blocked by auxiliary filter but not used for template generation; the other 100 spam tweets are blocked by template matching or noise detection, or caught by auxiliary filter and used for template generation.

Another trickier case is when the attacker exploits Tangram to block future tweets that have no similar ones cached yet. In this case, we need to arm the auxiliary spam filter with more sophisticated semantic analysis. For example, although the attacker sends a large volume of tweets, all the tweets may not contain malicious content. Then again, the auxiliary spam filter suspends the spammer and the corresponding tweets without using them for template generation. Moreover, when the topic of the attacker's targeted "future tweets" becomes heated, honest users usually do not follow certain templates to compose tweets. Even if we extract a template from the attacker's tweets, it might not affect honest users that much.

## VII. Related Work

**Spam Detection.** *Judo* [10] also infers the underlying template used to generate spam. However, *Judo* (as well as its adaptation to web spam [11]) assume the presence of invariant substring in the template, and requires a clean spam trace instantiating the same template as input. These requirements are hard to satisfy in the OSN environment.

Researchers have proposed other approaches that fight spam using the textual content, including using the syntactical textual similarity within the same campaign [3], [31] and extracting signature of embedded URLs [16]. Meanwhile, other features of spam/spammers are used to fight spam as well. Egele et al. model account profiles and use anomaly detection to identify compromised accounts in OSNs [17]. Song et al. propose to use the social graph property to detect spam tweets [18]. Yang et al. use sender profile features among others [12]. Other proposed techniques include focusing on embedded URL information like redirection chains, DNS and WHOIS information and so on [4], [32], [33], classifying URLs' landing pages [5], [34] and using sender's reputation [13]–[15], [35]. Building sender profile features takes time and it is difficult to adopt for real-time detection. Few existing works can both do real-time detection and filter spam without URLs.

**Spam Measurement.** Thomas et al. examine a large corpus of suspended Twitter accounts in [6], which provides rich knowledge on Twitter spammers that inspires our work from multiple aspects. Cao et al. design and implement a malicious account detection system called SynchroTrap which was able to reveal a lot of malicious accounts [36]. Yang et al. design some more robust features to detect more Twitter spammers through in-depth analysis of the evasion strategies utilized by up-to-date Twitter spammers [37]. In addition, Grier et al. and Gao et al. discovered the popularity of compromised spamming accounts in Twitter and Facebook, respectively [2], [22]. Due to the different data collection method, most spamming accounts in our dataset are created by spammers. Yang et al. analyze the social network formed by spamming accounts and reveal different categories of legitimate accounts that follow spamming accounts [38]. Levchendo et al. and Kanich et al. study the monetization of spam campaigns [39], [40].

**Signature Generation.** The problem of spam template generation bears similarity with polymorphic worm signature generation [41], [42]. The worm signature generation is based on the assumption that polymorphic worm content contains invariant substrings, which is reasonable because some invariant bytes are crucial for successfully exploiting the vulnerability. However, this assumption is not solid in the context of spam detection, where spammers can express the same message using different expressions in human language. Our Twitter spam analysis supports this argument.

## VIII. Conclusion

We have proposed and evaluated Tangram, a template-based system for accurate and fast OSN spam detection. Our measurement study reveals that the majority of Twitter spam is likely to instantiate underlying templates. Based on the empirical findings, Tangram mainly employs template generation/matching to mitigate OSN spam. Tangram distinguishes from existing template generation work in that it can construct template in the absence of invariant substrings. Tangram detects OSN spam in real-time without a separate training phase. We further study several situational-awareness inference of spammers' strategy. The findings promise more features for detecting spam and spamming accounts.

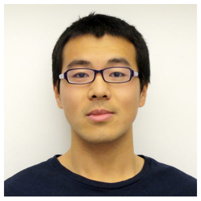
## References

[1] H. Gao, Y. Yang, K. Bu, Y. Chen, D. Downey, K. Lee, and A. Choudhary, "Spam ain't as diverse as it seems: Throttling osn spam with templates underneath," in *ACSAC*, 2014, pp. 76–85.
[2] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, and B. Y. Zhao, "Detecting and characterizing social spam campaigns," in *IMC*, 2010, pp. 35–47.
[3] H. Gao, Y. Chen, K. Lee, D. Palsetia, and A. Choudhary, "Towards Online Spam Filtering in Social Networks," in *NDSS*, 2012.
[4] S. Lee and J. Kim, "WarningBird: Detecting suspicious URLs in Twitter stream," in *NDSS*, 2012.
[5] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song, "Design and Evaluation of a Real-Time URL Spam Filtering Service," in *S&P*, 2011.
[6] K. Thomas, C. Grier, V. Paxson, and D. Song, "Suspended Accounts In Retrospect: An Analysis of Twitter Spam," in *IMC*, 2011, pp. 243–258.
[7] "Twitter acknowledges 23 million active users are actually bots," http://tinyurl.com/l755bvm.
[8] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage, "Spamcraft: An inside look at spam campaign orchestration," in *LEET*, 2009.
[9] ——, "On the spam campaign trail," in *LEET*, vol. 8, 2008, pp. 1–9.
[10] A. Pitsillidis, K. Levchenko, C. Kreibich, C. Kanich, G. Voelker, V. Paxson, N. Weaver, and S. Savage, " Botnet Judo: Fighting Spam with Itself ," in *NDSS*, 2010.
[11] Q. Zhang, D. Y. Wang, and G. M. Voelker, "Dspin: Detecting automatically spun content on the web," in *NDSS*, 2014.
[12] C. Yang, R. Harkreader, and G. Gu, "Die free or live hard? empirical evaluation and new design for fighting evolving twitter spammers," in *RAID*, 2011, pp. 318–337.
[13] G. Stringhini, C. Kruegel, and G. Vigna, "Detecting spammers on social networks," in *ACSAC*, 2010, pp. 1–9.
[14] F. Benevenuto, G. Magno, T. Rodrigues, and V. Almeida, "Detecting spammers on Twitter," in *CEAS*, vol. 6, 2010, p. 12.
[15] A. H. Wang, "Don't follow me - spam detection in twitter." in *SECRYPT*, S. K. Katsikas and P. Samarati, Eds. SciTePress, 2010.
[16] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov, "Spamming botnets: signatures and characteristics," in *Proc. of SIGCOMM*, vol. 38, no. 4, 2008, pp. 171–182.
[17] M. Egele, G. Stringhini, C. Kruegel, and G. Vigna, "COMPA: Detecting Compromised Accounts on Social Networks," in *NDSS*, 2013.
[18] J. Song, S. Lee, and J. Kim, "Spam filtering in twitter using sender-receiver relationship," in *RAID*, 2011, pp. 301–317.
[19] "What the trend," http://www.whatthetrend.com/.
[20] A. Ritter, S. Clark, O. Etzioni *et al.*, "Named entity recognition in tweets: an experimental study," in *EMNLP*, 2011.

[21] T. Jiang and M. Li, "On the approximation of shortest common supersequences and longest common subsequences," in *ICALP*, 1994.
[22] C. Grier, K. Thomas, V. Paxson, and M. Zhang, "@spam: the underground on 140 characters or less," in *CCS*, 2010, pp. 27–37.
[23] J. D. Lafferty, A. McCallum, and F. C. N. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," in *ICML*, 2001.
[24] "UNLEASH THE POWER OF THE LINK," https://bitly.com/.
[25] "Virustotal," https://www.virustotal.com/.
[26] "Wepawet," https://wepawet.iseclab.org/.
[27] "McAfee SiteAdvisor," http://www.siteadvisor.com/.
[28] "Safe Browsing Tool — WOT (Web of Trust)," http://www.mywot.com/.
[29] B. Stone-Gross, T. Holz, G. Stringhini, and G. Vigna, "The underground economy of spam: a botmaster's perspective of coordinating large-scale spam campaigns," in *LEET*, 2011.
[30] X. Jin, C. Lin, J. Luo, and J. Han, "A data mining-based spam detection system for social media networks," *Proceedings of the VLDB Endowment*, vol. 4, no. 12, 2011.
[31] L. Zhuang, J. Dunagan, D. R. Simon, H. J. Wang, and J. D. Tygar, "Characterizing botnets from email spam records," in *LEET*, 2008.
[32] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond blacklists: learning to detect malicious web sites from suspicious urls," in *KDD*, 2009.
[33] ——, "Identifying suspicious urls: an application of large-scale online learning," in *ICML*, 2009, pp. 681–688.
[34] D. S. Anderson, C. Fleizach, S. Savage, and G. M. Voelker, "Spamscatter: characterizing internet scam hosting infrastructure," in *USENIX Security*, 2007.
[35] S. Hao, N. A. Syed, N. Feamster, A. G. Gray, and S. Krasser, "Detecting spammers with snare: spatio-temporal network-level automatic reputation engine," in *USENIX Security*, vol. 9, 2009.
[36] Q. Cao, X. Yang, J. Yu, and C. Palow, "Uncovering large groups of active malicious accounts in online social networks," in *CCS*, 2014.
[37] C. Yang, R. Harkreader, and G. Gu, "Empirical evaluation and new design for fighting evolving twitter spammers," *Information Forensics and Security, IEEE Transactions on*, vol. 8, no. 8, pp. 1280–1293, 2013.
[38] C. Yang, R. Harkreader, J. Zhang, S. Shin, and G. Gu, "Analyzing spammers' social networks for fun and profit: a case study of cyber criminal ecosystem on twitter," in *WWW*, 2012, pp. 71–80.
[39] K. Levchenko, N. Chachra, B. Enright, M. Felegyhazi, C. Grier, T. Halvorson, C. Kanich, C. Kreibich, H. Liu, D. McCoy, A. Pitsillidis, N. Weaver, V. Paxson, G. M. Voelker, and S. Savage, "Click Trajectories: End-to-End Analysis of the Spam Value Chain," in *S&P*, 2011.
[40] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage, "Spamalytics: An empirical analysis of spam marketing conversion," in *CCS*, 2008, pp. 3–14.
[41] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez, "Hamsa: Fast signature generation for zero-day polymorphicworms with provable attack resilience," in *S&P*, 2006.
[42] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically generating signatures for polymorphic worms," in *S&P*, 2005.

**Tiantian Zhu** received his B.Eng. degree in information security from Northeastern University, Shenyang, China, in 2014. He is currently pursuing the Ph.D. degree with the College of Computer Science and Technology at Zhejiang University, Hangzhou, China. His research interests include OSN security and mobile security.

**Hongyu Gao** received the Ph.D. degree in computer science from Northwestern University, Evanston, IL, USA, in 2013. He is an research scientist at Shape Security, Inc. His research interests include computer networks and security, with a focus on detecting and preventing automation and fraudulent activity on the internet.

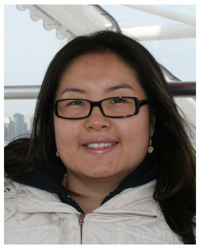


**Yi Yang** received his B.S. degree in computer science from University of Science and Technology of China, Hefei, China in 2009, and the Ph.D. degree in computer science from Northwestern University, Evanston, IL, USA, in 2015. His research areas include natural language processing, machine learning and business intelligence.

**Kai Bu** received the B.Sc. and M.Sc. degrees in computer science from the Nanjing University of Posts and Telecommunications, Nanjing, China, in 2006 and 2009, respectively, and the Ph.D. degree in computer science from Hong Kong Polytechnic University, Kowloon, Hong Kong, in 2013. Currently, he is an Assistant Professor with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China. His research interests include wireless networking and network security. He is a Member of the ACM, the IEEE, and CCF. He is a recipient of the Best Paper Award of IEEE/IFIP EUC 2011.

**Yan Chen** received the Ph.D. degree in computer science from the University of California, Berkeley, CA, USA, in 2003. He is a Professor with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL, USA. Based on Google Scholar, his papers have been cited over 7000 times and his h-index is 34. His research interests include network security, measurement, and diagnosis for large-scale networks and distributed systems. Prof. Chen won the Department of Energy (DoE) Early CAREER Award in 2005, the Department of Defense (DoD) Young Investigator Award in 2007, and the Best Paper nomination in ACM SIGCOMM 2010.

**Doug Downey** is an associate professor at Northwestern University in Electrical Engineering and Computer Science. His research focuses on natural language processing, machine learning, and artificial intelligence, with a particular focus on information extraction from Web content.



**Kathy Lee** is a Ph.D. student under the supervision of Prof. Alok Choudhary, in the Department of Electrical Engineering and Computer Science at Northwestern University. She received her bachelors degree in Computer Science from McGill University, Montreal, Canada.

**Alok N. Choudhary** received his PhD in electrical and computer engineering from the University of Illinois, Urbana-Champaign, in 1989. Since 2000, he has been a Professor at Northwestern University, Evanston, IL. Prof. Choudhary is a fellow of IEEE, ACM, and AAAS.