# Spam ain't as Diverse as It Seems: Throttling OSN Spam with Templates Underneath

Hongyu Gao⋆, Yi Yang⋆, Kai Bu*, Yan Chen⋆, Doug Downey⋆, Kathy Lee⋆, Alok Choudhary⋆
⋆Department of Electrical Engineering and Computer Science, Northwestern University
*College of Computer Science and Technology, Zhejiang University

## ABSTRACT

In online social networks (OSNs), 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 (63.0%) of the collected spam is generated with underlying templates. We therefore propose extracting templates of spam detected by existing methods and then matching messages against the templates toward accurate and fast spam detection. We implement this insight through *Tangram*, an OSN spam filtering system that performs online inspection on the stream of user-generated messages. Tangram automatically divides OSN spam into segments and uses the segments to construct templates to filter future spam. Experimental results show that Tangram is highly accurate and can rapidly generate templates to throttle newly emerged campaigns. Specifically, Tangram detects the most prevalent template-based spam with 95.7% true positive rate, whereas the existing template generation approach detects only 32.3%. The integration of Tangram and its auxiliary spam filter achieves an overall accuracy of 85.4% true positive rate and 0.33% false positive rate.

## Categories and Subject Descriptors

J.4 [**Computer Applications**]: Social and behavioral sciences

## General Terms

Human Factors, Measurement, Security

## Keywords

Online social networks, spam, spam campaigns

## 1. INTRODUCTION

Spam has been plaguing the Internet community for more than a decade. With the tremendous popularity of online social networks (OSNs) in recent years, spammers quickly start to exploit this new media channel. Despite the development of countermeasures, spammers find their way to adapt and stick. Research reveals that on Twitter, one of the most popular OSNs nowadays, more than 4% of collected tweets are spam [27], which has slipped through all the deployed defense mechanisms. Researchers have proposed to combat OSN spam from multiple angles, including mining the textual content [6], studying the redirection chains of embedded URLs [15] as well as classifying the URL landing pages [26]. Despite the effort to build OSN spam mitigation systems, it is not yet clear what techniques the spammers are using to construct OSN spam, and how the techniques evolve. This missing piece of information plays a crucial role to inspire effective designs to throttle OSN spam.

In this paper, our first contribution is to reveal the OSN spam generation techniques according to spam textual patterns (**Section 2**). We conduct a measurement study, identifying 115 campaigns from more than 500 thousand spam messages. We find that the majority of spam (63.0%) is generated with underlying templates, which is consistent with prior email spam research [12, 13, 22]. Templates are valuable for spammers, because they let spammers control and customize the semantic meaning of generated messages to boost the conversion rate. Meanwhile, they generate diversified messages that are more difficult to detect. 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 addition, the spam without underlying templates calls for effective countermeasures as well.

In particular, we identify three major challenges that render existing spam template generation techniques ineffective.

1. **Absence of invariant substring in template.** Prior spam template generation research [22,33] 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.
2. **Prevalence of noise.** Spammers extensively add semantically unrelated noise words into spam messages. The presence of noise diversifies spam messages, and increases the difficulty to identify semantically meaningful text segments.

3. **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.

Our second contribution is to propose Tangram, a system that performs effective template generation to combat OSN spam (**Section 3**). Many existing methods detect spammers instead of spam. These methods are based on account activity and need long observation periods for the account features to accumulate [3, 25, 29, 31]. Many other detection approaches are based on URL analysis, which inherently cannot detect spam without URLs [15, 26, 30]. Researchers have revealed significant amount of such spam [4]. The few existing methods that detect spam with or without URLs in real-time suffer from high false positive rates [4, 6, 24]. In contrast, Tangram is the first accurate online OSN spam detection system that detects spam with or without URLs. It extracts templates of spam detected by existing methods and then matches messages against the templates toward accurate and fast spam detection. In practice, it can be deployed in combination with heterogeneous detection methods, like account and URL analysis, to optimize detection accuracy. Specifically, Tangram is highly accurate because of the following four innovations.

1. **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, which are used to match future spam.
2. **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 are used to generate templates.
3. **Break spam heterogeneity.** We pre-cluster spam and perform template generation within individual partitions. The template generation procedure also discards outlier messages in the partition.
4. **Build a double defense.** We mitigate spam without underlying templates using a supplementary module that detects spam with excessive semantically unrelated noise words. In an adversarial setting, spammers may deliberately copy or adapt benign content copied from other sources in attempt to avoid detection by Tangram. Since spammers need to use popular content to attract the audience and they generate a large number of spam, the OSN administrator will observe the popular content with replaced URLs in large volume, which defeats the spammers' purpose to evade detection. In addition, we can equip Tangram with multiple heterogeneous detection modules in practice. It is hard for spammers to evade detection from multiple angles simultaneously.

We evaluate Tangram's ability to detect real-time spam on a large Twitter dataset of 17 million tweets (**Section 4**). Equiped with one necessary auxiliary spam filter, Tangram detects 95.7% of the most prevalent template-based spam. The complete system detects Twitter spam with 85.4% true positive rate and 0.33% false positive rate. The true positive rate is similar with existing systems [4, 6, 24], while the false positive rate is at least 2 times better than that in [6] and 10 times better than [4, 24]. Tangram reacts to newly emerged spam campaigns quickly. Finally, we investigate Tangram's robustness (**Section 5**) and related work (**Section 6**).

## 2. 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. We find that the majority of spam is generated by underlying templates. This finding motivates Tangram.

### 2.1 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. In the data collection process, we continuously downloaded popular Twitter Hashtags from the website *What the Trend* [1]. 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. We, however, would like to emphasize that the numerical accountIDs in our dataset followed a uniform distribution, 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, showing that our collected spam had few duplicates. The other tweets were labeled as legitimate. Using the same method, we collected another relatively small tweet dataset generated during January, 2012, containing 46,844 spam tweets. 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].

| | | |
|---|---|---|
| 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* |

**Table 1: Retrofitted sample spam from a template-based campaign. (*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.*)**

### 2.2 Template-based Spam Dominates

Through mining the textual pattern of spam, we find that the majority (63.0%) of the 2011 spam dataset shares under-

[1]The spam labeling approach assumed that suspended accounts generated no legitimate message. To validate this assumption, we randomly selected 100 suspended accounts and manually checked all their posted tweets, which were indeed spam. Although spamming was not the only reason for account suspension in Twitter, our manual checking showed that it was the overwhelmingly dominant reason.

lying templates. Table 1 shows five (intentionally retrofitted) sample spam tweets from a much larger, but 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.

**Spam Template Model.** 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 (e.g., celebrity name in Table 1). 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. *Dictionary* macros carry semantic meanings so that spammers can lure the recipients to visit the embedded URLs. 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. A noise macro can be implemented by, for example, randomly generating a character string, mentioning Twitter users, etc.; it is not our interest to differentiate among them. 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 [22, 33] that relies on invariant substrings.

## 2.3 Template-based Spam Continues Dominating

**Spam Categorization.** 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. In contrast with template-based category, *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, e.g., "browsing statistics click Celebrity John $URL$" and "interesting site on Celeb J $URL$". In addition, semantically dissimilar spam consists of *No-content Spam* that contains very little semantically meaningful textual content and *Other Spam* that we have not systematically categorized.

**Template-based spam continues dominating other categories of spam.** Table 2 provides the popularity of four spam categories in June/July, 2011 and January, 2012. Template-based spam remains to be the most popular category in 2012, with its percentage increasing to 68%. The no-content category almost vanishes. Its percentage dramatically drops to 0.3%. It is possible that the no-content category exhibits strong patterns and can be easily blocked. The

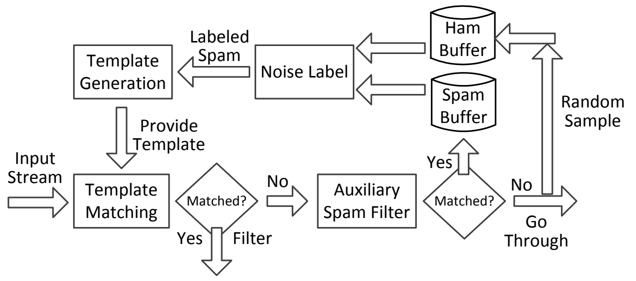

**Figure 1: Tangram framework: The template generation and matching overview.**

increasingly popular template-based spam indicates that our detection method with focus on spam template generation is effective to combat modern OSN spam.

| Spam Category | | 2011 | 2012 |
|---|---|---|---|
| Similar Semantic | Template-based | 63.0% | 68.3% |
| | Paraphrase | 14.7% | 12.9% |
| Dissimilar Semantic | No-content | 8.4% | 0.3% |
| | Others | 13.9% | 18.5% |

**Table 2: The popularity of four spam categories in June/July, 2011 and January, 2012, respectively.**

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

In this section, we present Tangram, a template-based spam detection system with high accuracy and speed. We first formulate the notions of template, template matching and template generation. Next, we present the details of the online Tangram system. Tangram does not need the knowledge of spam categorization (Section 2.3) as input. It only needs the stream of raw messages, plus a small number of labeled tweets for noise labeling as described in Section 3.4.

## 3.1 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. We further discuss the dependence on the auxiliary spam filter in Section 5.

**Template Matching and Template Generation.** We define a *template* to be a sequence of macros of two types, dictionary and noise (Section 2.2). 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 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. The system input is a mixture of spam instantiating different templates and spam *without* underlying templates. This differentiates our system from previous spam template generation work [22].

The template generation first identifies the subset of spam messages believed to share the same template (Section 3.3). These messages are tokenized into sequences of words. After executing noise detection (Section 3.4), 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[2]. 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). We develop approximate techniques that work well in practice, as described below.

## 3.2 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 1 as a running example to elaborate our approach. We expand the approach to generate templates on a mixture of spam instantiating multiple templates in Section 3.3.

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 us infer the correspondence of segments between messages. For example, if we observe the first two messages in Table 1 from a campaign, 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 -".

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). We use an approximation algorithm named Majority-Merge [10] because of its simplicity. It takes $n$ sequences as input, and initializes the supersequence, $s$, as an empty string. Next, it iteratively chooses the majority of the leftmost tokens of the input sequences, denoted as $a$, and appends $a$ to $s$. Meanwhile, the leftmost $a$ are deleted from the input sequences. It repeats this step until all sequences are empty, and outputs $s$. Each token in the output supersequence is trivially a substring shared by some subset of the campaign. Desirable substrings are *i*) shared by large subsets and *ii*) long. We achieve goal *i*) by producing a shorter supersequence.

We build a matrix during the execution of Majority-Merge algorithm. Table 3 shows an excerpt (due to page width limit) of such a table for the example campaign. The header row is the supersequence output by the Majority-Merge algorithm. The rest of the rows represent the input sequences, one row for each sequence. We label the $j_{th}$ column using the token that is chosen in the $j_{th}$ step of the Majority-Merge algorithm. The cell at row $i$, column $j$ will be assigned the column label if the $i_{th}$ sequence is picked in step $j$. Otherwise it will be an empty string, $\varepsilon$. Naturally, the concatenation of each row is exactly the corresponding input sequence. We denote this property as the *supersequence property*.

To produce a shorter supersequence, we need to merge columns that shares the same label, while maintaining the supersequence property. After merging, the cell will be assigned the column label if either cell before merging has been assigned so. 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). 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 4 excerpts 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 4 are

[2]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 [23] and 2) our segments often do not correspond to typical NLP segments such as noun phrases.

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

**Table 3: Excerpt of 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 | - | RIP | Celeb | C | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Big | Name | A | $\varepsilon$ | $\varepsilon$ | an | eye-catching | action | - | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | ... |
| $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | Celebrity | B | an | eye-catching | action | - | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | ... |
| Big | Name | A | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | ... |
| $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | Celebrity | B | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | ... |
| $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | RIP | Celeb | C | ... |

**Table 4: Excerpt of the intermediate matrix after the matrix column reduction step. The header row is a shorter supersequence.**

concatenated because "Big" always maps to "Name", but the 5th and the 6th columns in Table 4 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, which helps reveal the true template by assembling tokens (words) into word phrases. For example, the three separate columns "Big", "Name", and "A" in Table 4 become one celebrity name. Second, the cells on the same column after column concatenation may have different contents. 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 generated regular expression representation from Table 4 is ˆ(Big Name A | Celebrity B | RIP Celeb C) (offensive content , look at this video | an eye-catching content - ) URL$, where "ˆ" and "$" respectively mark the beginning and the ending of a message. The output regular expression matches all input sequences that build this matrix.

## 3.3 Multi-campaign Template Generation

We now expand single campaign template generation to multi-campaign template generation over spam instantiating *different* templates or even without underlying templates. An intuitive solution is to first separate the spam into distinct campaigns automatically, then invoke the single campaign template generation procedure on each one of them. We use a clustering and refining approach to this task.

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, before applying the system on the much larger dataset. 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 then refine the clusters using the single campaign template generation algorithm. Intuitively, spams from different campaigns will result in non-compact templates, a fact we utilize to identify which spam should be removed from a given cluster. Specifically, during template generation, we repeatedly remove the row with the largest number of $\varepsilon$, because it likely results from a clustering error. A column will be removed if all its non-$\varepsilon$ cells are removed. We repeat this process until the matrix contains sufficiently low number of $\varepsilon$.

## 3.4 Noise Labeling

Spam tweets often mention other users, popular terms and hashtags that are unrelated to the semantics of the rest of the tweet. Earlier research has confirmed this phenomenon [8]. Such content helps to expose spam to a larger audience, because users may search or browse tweets by topic. It also diversifies spam and increases the difficulty to detect spam. 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 approach, Conditional Random Fields (CRFs) [14]. 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. As a result, 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 (i.e. using @).

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.

## 4. EXPERIMENTS

We evaluate Tangram using the labeled dataset in Section 2.1 as ground truth. The two major metrics that we use to evaluate the system are accuracy and speed. We conduct a strict accuracy evaluation. We only count spam caught by template matching or noise detection as true positives. We count spam missed by these two but caught by the auxiliary spam filter as false negatives. In this way, we are only evaluating the detection accuracy of the modules proposed in this paper, not the accuracy of the auxiliary spam filter. 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. We leverage an existing online OSN spam filtering tool [6] for this task to conduct 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.

### 4.1 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 spam messages that are not template-based, 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 the TP rate of template-based messages. The overall TP and FP rate are 76.2% and 0.12%, respectively.

**True Positive Analysis.** Table 5 reports a detailed breakdown of true positive rate into different spam categories. Tangram has two detection modules. Both modules perform 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. The noise detection module can detect no-content spam with 73.8% TP rate. Unfortunately, the true positive rate of the other two spam categories is lower. About 80% of the false negatives (spam misclassified as legitimate) 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 2.1), 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. The comparison approach exploits characteristics that may only apply to our specific dataset. Hence, we only 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

| 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% |

**Table 5: The detection accuracy of Tangram and two existing systems compared in Section 4.2. The last column shows the accuracy if we combine Tangram and the syntactical approach.**

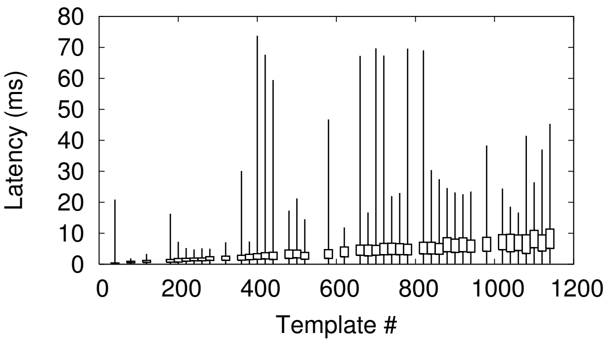

**Figure 2: The box plot of template matching latency as a function of the number of generated templates.**

the auxiliary filter, so templates are generated to match them.

## 4.2 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 [6] (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 [6]. 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 5 lists the detailed accuracy comparison. The difference between the spam detected by these two systems indicates that they can potentially complement each other. Both systems work in a similar way and it is straightforward to integrate them. For example, a message can be labeled as spam if either system blocks it. This simple integration suffers from the increased FP rate of 0.33%, but can boost the 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* [22]. *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 5, column “Judo”. Different from our system, *Judo* requires training set that contains *pure spam* generated by the *same* underlying template. As a result, in real-world deployment the system relies on the assumption that only a small number of templates are actively used at a given time, so that the training set is pure at least within a given small time window. We implement the template generation mechanism of *Judo* as described in [22], 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. In comparison, our system achieves much higher accuracy on the same corpus.

## 4.3 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 during our online experiment. 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 the time needed for template generation is larger than the time for template matching, the template generation time is *not* the bottleneck of Tangram, since template generation is performed in parallel with template matching.

## 4.4 Sensitivity for New Campaigns

We take the five largest campaigns, one of which matches the template instantiated by spam in Table 1, 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

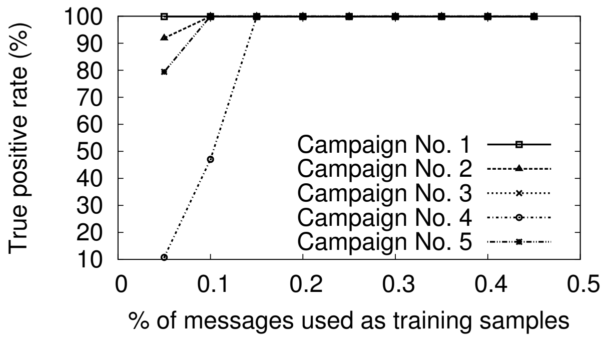

**Figure 3: The TP rate when template generation is performed separately for each campaign, varying the size of training set. Most observation points reach 100% TP rate.**

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.

## 5. DISCUSSIONS

**How to increase the attack resilience of the auxiliary spam filter?** 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. Spammers can try to evade Tangram by evading the auxiliary spam filter, so that no training samples are available for template generation. We can address this by introducing heterogeneity into the auxiliary filter. Using a combination of multiple inherently different existing spam detection systems as the auxiliary filter makes it very hard for spammers to evade *all* of them.

**How to mitigate training sample poisoning?** Powerful adversaries can manipulate the training samples to mislead the training of the detection system. One possible way is to inject popular legitimate content into spam, hoping that the generated template matches a large number of legitimate messages. However, our experimental results show that Tangram will not generate template for spam with seemingly legitimate content. As a second precaution, we can set a threshold and only deploy the templates that incur a false positive rate lower than the threshold.

Another popular attack is to send spam with different patterns from the training samples after the detection system finishing training. Tangram is inherently immune to such attack, because it does not have separate training and testing phase. If spammers produce spam from a non-stationary distribution, Tangram can also detect it as long as sufficient large amount of spam is produced from each distribution.

**How to mitigate paraphrase spam?** The major difficulty for Tangram to detect paraphrase spam is the lack of ordering among semantically meaningful segments, which is assumed by the template model. Nonetheless, semantic analysis techniques that do not consider the word ordering may be used to mitigate such spam, e.g., clustering based on cosine similarity using bag-of-words model. In addition, paraphrase generation is still an active area of research in Natural Language Processing [20]. So is paraphrase detection [5]. The detection of paraphrase spam can leverage existing paraphrase detection approaches, and is one of our future research directions.

**How to mitigate spam that re-uses legitimate content?** Since spammers need to use popular content to attract the audience and generate a large number of spam, the OSN administrator will observe the popular content with replaced URLs in large volume, which defeats the spammers' purpose to evade detection. Also, in practice, we can equip Tangram with multiple heterogeneous detection modules that do not rely on spam content.

## 6. RELATED WORK

**Spam Detection.** *Judo* [22] also infers the underlying template used to generate spam. However, *Judo* (as well as its adaptation to web spam [33]) 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. We also show via experiments that the *Judo* design does not yield satisfactory detection accuracy on a real-world OSN trace. In contrast, our system is designed specifically for OSN and is not limited by the above two requirements.

Researchers have proposed other approaches that fight spam using the textual content, including using the syntactical textual similarity within the same campaign [6,34] and extracting signature of embedded URLs [30]. 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 [4]. Song et al. propose to use the social graph property to detect spam tweets [24]. Yang et al. use sender profile features among others [31]. Other proposed techniques include focusing on embedded URL information like redirection chains, DNS and WHOIS information and so on [15,18,19], classifying URLs' landing pages [2,26] and using sender's reputation [3,9,25,29]. Building sender profile features takes time and it is difficult to adopt for real-time detection. Compared with our work, although some of the above approaches report higher spam coverage, they incur significantly higher false positive rate, showing that the features they use are less precise than spam templates. More importantly, very few existing works can do real-time detection and filter spam without URLs simultaneously, where as Tangram by design can achieve both.

**Spam Measurement.** Thomas et al. examine a large corpus of suspended Twitter accounts in [27], which provides rich knowledge on Twitter spammers that inspires our work from multiple aspects. In addition, Grier et al. and Gao et al. discovered the popularity of compromised spamming accounts in Twitter and Facebook, respectively [7, 8]. 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 [32]. Levchendo et al. and Kanich et al. study the monetization of spam campaigns [11, 16]. In comparison, our work focuses on detecting spam, whereas the above works focus on revealing spammers' characteristics using known spam.

**Signature Generation.** The problem of spam template generation bears similarity with polymorphic worm signature generation [17, 21]. 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. Venkataraman et al. formalize the limits on the pattern-extraction algorithms for signature generation in the presence of powerful adversaries [28]. Due to the difference in the underlying assumptions, i.e., with and without invariant substrings, it is hard to directly apply their conclusions to our spam template generation problem.

## 7. CONCLUSION

We have proposed and evaluated Tangram, a template-based system for accurate and fast OSN spam detection. Our measurement study reveals that 63% 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. Experimental results show that Tangram can detect 95.7% of the most prevalent template-based spam in the collected Twitter dataset. Equipped with one necessary auxiliary spam filter, the combined system achieves an overall true positive rate of 85.4% and a false positive rate of 0.33%.

## Acknowledgement

This work is supported in part by the following grants: NSF awards CCF-1029166, ACI-1144061, IIS-1343639, and CCF-1409601; DOE award DESC0007456; DARPA contract D11AP00268; Fundamental Research Funds for the Central Universities under Grant No. 2014QNA5012. We would also like to sincerely thank ACSAC 2014 chairs and reviewers for their helpful feedback.

## 8. REFERENCES

[1] What the trend. `http://www.whatthetrend.com/`.

[2] D. S. Anderson, C. Fleizach, S. Savage, and G. M. Voelker. Spamscatter: characterizing internet scam hosting infrastructure. In *USENIX Security*, 2007.

[3] F. Benevenuto, G. Magno, T. Rodrigues, and V. Almeida. Detecting spammers on Twitter. In *CEAS*, 2010.

[4] M. Egele, G. Stringhini, C. Kruegel, and G. Vigna. COMPA: Detecting Compromised Accounts on Social Networks. In *NDSS*, 2013.

[5] S. Fernando and M. Stevenson. A semantic similarity approach to paraphrase detection. In *CLUK*, 2008.

[6] H. Gao, Y. Chen, K. Lee, D. Palsetia, and A. Choudhary. Towards Online Spam Filtering in Social Networks. In *NDSS*, 2012.

[7] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, and B. Y. Zhao. Detecting and characterizing social spam campaigns. In *IMC*, 2010.

[8] C. Grier, K. Thomas, V. Paxson, and M. Zhang. @spam: the underground on 140 characters or less. In *CCS*, 2010.

[9] 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*, 2009.

[10] T. Jiang and M. Li. On the approximation of shortest common supersequences and longest common subsequences. In *ICALP*, 1994.

[11] 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.

[12] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. On the spam campaign trail. In *LEET*, 2008.

[13] 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.

[14] J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML*, 2001.

[15] S. Lee and J. Kim. WarningBird: Detecting suspicious URLs in Twitter stream. In *NDSS*, 2012.

[16] 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.

[17] 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.

[18] 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.

[19] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Identifying suspicious urls: an application of large-scale online learning. In *ICML*, 2009.

[20] N. Madnani and B. Dorr. Generating phrasal and sentential paraphrases: A survey of data-driven methods. *Computational Linguistics*, 36(3):341–387, 2010.

[21] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *S&P*, 2005.

[22] 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.

[23] A. Ritter, S. Clark, O. Etzioni, et al. Named entity

recognition in tweets: an experimental study. In *EMNLP*, 2011.

[24] J. Song, S. Lee, and J. Kim. Spam filtering in twitter using sender-receiver relationship. In *RAID*, 2011.

[25] G. Stringhini, C. Kruegel, and G. Vigna. Detecting spammers on social networks. In *ACSAC*, 2010.

[26] 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.

[27] K. Thomas, C. Grier, V. Paxson, and D. Song. Suspended Accounts In Retrospect: An Analysis of Twitter Spam. In *IMC*, 2011.

[28] S. Venkataraman, A. Blum, and D. Song. Limits of learning-based signature generation with adversaries. In *NDSS*, 2008.

[29] A. H. Wang. Don't follow me - spam detection in twitter. In S. K. Katsikas and P. Samarati, editors, *SECRYPT*, pages 142–151. SciTePress, 2010.

[30] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming botnets: signatures and characteristics. In *Proc. of SIGCOMM*, 2008.

[31] 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.

[32] 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.

[33] Q. Zhang, D. Y. Wang, and G. M. Voelker. Dspin: Detecting automatically spun content on the web. In *NDSS*, 2014.

[34] L. Zhuang, J. Dunagan, D. R. Simon, H. J. Wang, and J. D. Tygar. Characterizing botnets from email spam records. In *LEET*, 2008.

## APPENDIX

## A. TEMPLATE GENERATION COMPUTATIONAL CHALLENGE

We prove that the template generation problem is NP-hard. We use the following notions for ease of presentation:

1. $len(S)$**:** When $S$ is a token sequence, this operation computes the number of tokens in $S$. When $S$ is a set or a list, this operation recursively computes the sum of $len()$ for all the elements in $S$. $len(\varepsilon)$ is zero.
2. $cat(S)$**:** This operation concatenates the elements of a set $S$ in arbitrary order and returns the concatenation.
3. $[s_1, s_2, ..., s_k]$**:** An ordered list containing $k$ elements.

**Single Campaign Template Generation (SCTG):** We start with the problem to reconstruct the underlying template given a set of spam messages instantiating the *same* template.
**Input:** $n$ token sequences, instantiating the *same* template.
**Output:** An ordered list $\mathbf{L} = [S_1, S_2, ..., S_k]$, where each $S_i$ is a set of token sequences. $S_i$ may contain $\varepsilon$ (representing nothing). Every input token sequence can be represented as the concatenation of one element in each $S_i$. Meanwhile, $len(\mathbf{L})$ is minimized.
**Hardness: SCTG is NP-hard.** We prove this by reducing Shortest Common Supersequence Problem, a well-known NP-hard problem, into SCTG (Table 6).

```
Input: A set of $n$ token sequences $M = \{m_1, m_2, ..., m_n\}$
Reduction:
Find $\mathbf{L}$, the SCTG solution of $M$.
$S :=$“”
Foreach $S_i$ in $\mathbf{L}$:
  $S := S + cat(S_i)$
End Foreach
$S$ is the shortest common supersequence of $M$.
Proof: $S$ is the shortest common supersequence of $M$
```

It is trivially true that $S$ is a common supersequence of $M$.
Assume $\exists S'$, such that $S'$ is a common supersequence of $M$ and that $len(S') < len(S)$.
We can construct an ordered list $\mathbf{L}'$ of $len(S')$ elements, where the $i_{th}$ element is a set {the $i_{th}$ token in $S'$ , $\varepsilon$}.
Apparently, every $m_i \in M$ can be represented as the concatenation of one element of each set in $\mathbf{L}'$.
Because $\mathbf{L}$ is the output of the SCTG solver, we must have $len(\mathbf{L}) <= len(\mathbf{L}')$.
Because $len(\mathbf{L}) = len(S)$, $len(\mathbf{L}') = len(S')$ and $len(S') < len(S)$, we have $len(\mathbf{L}) > len(\mathbf{L}')$.
We thus reach a contradiction.
Hence we reject the assumption and prove that $S$ is the shortest common supersequence of $M$.

**Table 6: The reduction from Shortest Common Supersequence Problem to SCTG.**

**Multiple Campaign Template Generation (MCTG):** In real-world deployment scenarios, the system is expected to receive a mixture of spam instantiating multiple templates, and it is non-trivial to separate them. Accordingly, the system should be able to generate multiple templates from the input spam. Intuitively, the MCTG problem is at least as hard as the SCTG problem, since a MCTG solver is able to solve the SCTG problem as well.

## B. CORRECTNESS OF MATRIX COLUMN REDUCTION

In Section 3.2, we state three conditions to merge matrix columns. *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 $\varepsilon$. Under the three conditions, we prove that the *supersequence property* holds after merging by contradiction. Let column $j$ and column $k$ satisfy the above three conditions, and we assume that after merging column $k$ into column $j$, the resulting concatenation of column labels are not a supersequence of some input sequence $i$ (row $i$). There is at least one token in sequence $i$ that is not covered by the resulting concatenation of column labels. Let it be at the $m_{th}$ column in the original matrix. $m$ must be $k$ because other columns still perserve after merging. Hence, row $i$ column $k$ must not be $\varepsilon$. We denote it as $t$. Due to *condition ii*), row $i$ column $j$ must be $\varepsilon$. Due to *condition iii*), the subsequence of row $i$ between column $j$ and column $k$ is merely one token, $t$. Since column $j$ and column $k$ have identical token (*condition i*)), after merging the label of columns between $j$ and $k$ can cover $t$. Hence, the resulting column labels are still a supersequence of input sequence $i$. We hereby reach a contradiction, and the proof is complete.