Hinky: Defending Against Text-based Message Spam on Smartphones

Abdelkader Lahmadi, Laurent Delosière, Olivier Festor

Hinky: Defending Against Text-based Message Spam on Smartphones

Abdelkader Lahmadi
LORIA - Nancy University
Nancy, France
Email: abdelkader.lahmadi@loria.fr

Laurent Delosière
LORIA - Nancy University
Nancy, France
Email: delosieres@loria.fr

Olivier Festor
INRIA Nancy Grand Est
Nancy, France
Email: Olivier.festor@loria.fr

***Abstract*—We present a defense platform against text-based message spam on SmartPhones. We focus in particular on Short Message Service (SMS) based SPAM. Our solution relies on a social network based collaborative approach to filter this type of spam using Bloom filters and content hashing. We detail the design of the supporting framework and validate its efficiency in minimizing false positives and limiting the storage space. We show that a content hashing based approach provides better lookup than a bloom filter with the same storage space and false positive probability close to 10^{-9}.**

***Index Terms*—SMS, Spam, Collaborative, Filtering, Bloom Filters,**

I. Introduction

Text based messaging services are very popular where 500 millions of SMS were sent in France to celebrate the new year 2011. These services are also widely used on emerging social networks like Twitter or the basic SMS service on mobile phones. Nowadays, the cost of SMS is decreasing and many providers offer unlimited texting plans. This cost cut makes mobile-phone users a more attractive target for spammers. SMS spam is declined through different threats. Simple spam advertising is the most known threat, but malicious SMS can also carry malicious content like malware. Phishing or fraud SMS also jeopardize users privacy. SMS spam has long been considered as a minor problem, mostly because email spam is cheaper and mobile phones have limited communication capabilities to infect other users or browse phishing or advertisement web sites. However, modern SmartPhones offer more capabilities than traditional mobile phones. They are able to compute and communicate using different communication means: WiFi, UMTS and GSM. These devices also come with different applications which use the communication capabilities. Focusing on these communication capabilities, spammers are interested in delivering abusive or malicious SMS content to smartphone users.

In France for example, operators have been encouraging subscribers to report SMS spam since October 2008. Phone users can forward offending messages to the Stop-Spam service via the short code 33700. An initial response prompts them to reply with the spammer's phone number, with a final SMS acknowledging the completed spam report. The service has received almost half a million spam reports in this way. This has resulted in the disconnection of only 300 phone numbers for spamming, although many more have received cease-and-desist orders. The operators can't just block offending SMS messages or shuffle them into a spam folder as many e-mail service providers do. For one thing, the senders have paid for the messages, so the operators are contractually obliged to deliver them unless they can prove that the sender has breached their terms of service. Also, creating an SMS spam folder would mean updating the firmware on millions of phones. We believe that SMS spam has to be considered as a global problem where mobile spam is generally originating from senders outside the receivers mobile network operators.

Text based spam, mainly mail target spam has been extensively investigated [1], but little effort has been devoted to understanding their applicability on SMS spam. One of these defense techniques is the collaborative spam detection approach. In particular, this approach relies on exchanging black lists between users to block a detected spam on their own devices. This implies the usage of an efficient data structure to represent these black lists.

In this paper, we study a collaborative SMS spam filtering approach over smartphones using combined hashing and bloom filters techniques found in the literature. In section II we review the efficiency of these techniques regarding their false positive probabilities and storage space. Section III discusses the design of our defense solution against SMS spam using a collaborative approach and combining these techniques. We carried a series of experiments designed to test to what extent these techniques are effective in addressing SMS spam filtering. Section IV details the real implementation of the solution on Android based smartphones. Section V concludes with a summary of the contributions of this paper and presents future work.

II. Background and related work

A. Collaborative spam detection

A spam filtering mechanism can be built manually by the mobile's user in which he tags an SMS message as spam and where the SMS sender is introduced in a blacklist. Next time, an SMS is received from this sender it is treated as spam and the SMS is discarded. This approach is limited since each user does not benefit from other user's black lists. A collaborative approach is based on simple and powerful insights. When a user tags an SMS as spam, then this human effort is shared

among other users. When a message is identified as spam by somebody elsewhere, other users of the collaborative platform are protected from this undesired message. Our approach is similar to collaborative Anti-Spam services such as ALPACAS [2], [3], [1] in that it employs a reporting and querying functions to identify if an SMS is a spam. However, these solutions were provided for spam email and designed to run over servers with important resources and connectivity.

B. Filters representation

1) Bloom filters: Bloom filters [4] were designed as a space-efficient data structure for fast membership testing. A Bloom filter is a vector of `m` bits, each initially set to 0. The insertion of an element `x` in the filter requires to compute `k` hash functions. Each hash function provides a position in the range $\{0, 1, ..., m-1\}$ to be set to 1 on the filter. The query of an element `y` also requires to compute `k` hash functions and to construct a mask which contains the k positions set to 1. Then, a bitwise `AND` operation between the mask and the Bloom filter is performed. If the result equals the mask, then the answer is Yes, else the answer is No. The major drawback of Bloom filters is false positives. A false positive occurs when an element x is not in the filter but its different positions computed by k hash functions are all set to 1 due to overlap with other entries. The probability of false positive (fp) can be calculated as follows after the insertion of `n` elements:

$$fp = (1 - (1 - \frac{1}{m})^{nk})^k \quad (1)$$

The minimum of the probability of a false positive is reached when:

$$k = \frac{m}{n} \times ln(2) \quad (2)$$

In such case the minimum of false positive probability is estimated to $fp_{min} = (\frac{1}{2})^k$. In section III, we validate this estimation using simulation to identify its limits regarding the filter size and the number of hash functions. Many network applications have used Bloom filters for routing table lookup, packet classification and peer-to-peer networks [4]. Another major drawback of a Bloom filter is the difficulty of deleting inserted elements. A simple solution is the cold cache [5] where one removes all elements when the filter is full. The problem with this solution, is that no data remains in the filter and we loose useful information. In [5], authors propose the double buffering method for the deletion of old data. The Bloom filter is divided into sub filters: an active and a warm-up filter. The active filter stores all the recent inserted elements and the warm-up filter is always a subset of the active filter. In [6], the author proposes an enhancement of this approach through the use of two active sub filters. When one filter is full, the second is flushed and the two filters switch their roles. In our work, we adopted this approach for Bloom filter aging.

2) Hashing based filters: Instead of using a Bloom filter to query if a SMS is member of inserted spam messages in the filter, we may use a hashing based method. A hashing function is used to create a hashed value of the SMS reported as SPAM and insert it in a list. A first advantage of using a hashing method is to guaranty anonymity of the SMS content. The limit probability of a false positive within a list containing `n` hashed values is the probability to get at least a false positive. Therefore, we obtain:

$$fp = 1 - (1 - \frac{1}{2}^b)^n \quad (3)$$

Where b is the hashed value size.

III. Design of the Defense System

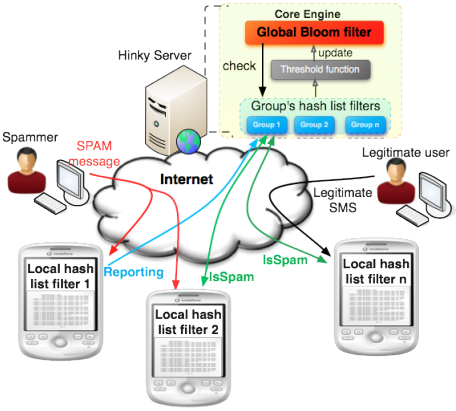

Fig. 1: Hinky platform architecture.

Figure 1 depicts the architecture of the Hinky platform. Our system relies on two components. The first component is located on the user's mobile phone. It contains a local filter maintained by the mobile user. The second component is the core engine. It is located on one or several collaborative servers. It contains the group filters shared by trustworthy users and a global filter shared by different groups.

A. Efficient Filtering Design

We first formulate the SMS filtering problem with both Bloom filters and hashing functions. We present the performance of each of them in terms of false positives, lookup time and storage size. The characteristics of mobile phones make the SMS filtering problem using signature matching unique: (i) the filtering system has to keep the spam signature as small as possible, because it needs to support a large number of spams. Storing undesired SMS PDUs is waste since the defense system's aim is to remove them and protect the user from their content. Moreover, we want to make a fast decision to match an incoming SMS against the set of available signatures instead of parsing the SMS and compare its content with the set of available SMSs. (ii) as stated before, the SMS service is critical and the removal of a legitimate SMS will be unacceptable. The filtering technique has to keep a null or a very low false positive rate; (iii) in a collaborative filtering approach, we have to handle multiple filters from different users. Thus, the raising issue is to merge them then perform matching or perform per-filter matching. (iv) Bloom filters were stated as a good candidate for filtering e-mail spam

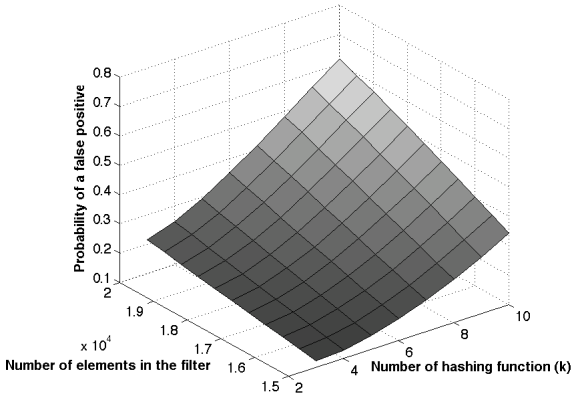

Fig. 2: The probability of a false positive in a Bloom filter of size 8KB.

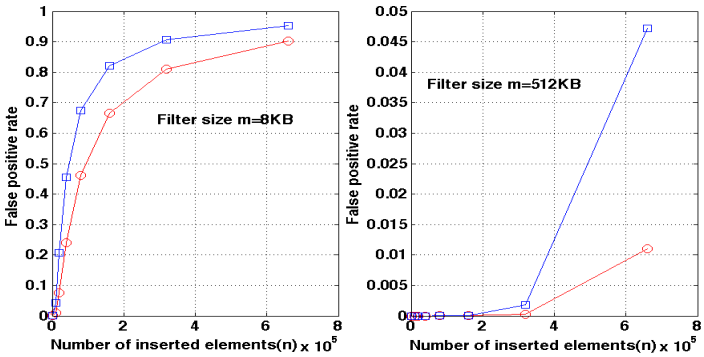

Fig. 3: Impact of the number of inserted elements on the false positive rate.

[3], however their performance compared to a hashing based signature for SMS is not assessed. In Hinky, architecture we are using different types of filters for local, group and global treatment. Each of them is different in terms of size and available resources on the hosting host. Choosing the right filter representation for each of them is our goal.

B. Bloom Filter numerical analysis

We have performed a numerical analysis of Bloom filters to identify their false positive probability and rate for filtering system level SMS messages.

1) Probability of a false positive: We are interested in the trade-off between the number of hash functions and the number of inserted elements in the best case where k is computed by the equation 2. We observe that k decreases when the number of inserted elements increases. However, when we increase the size of the filter, we need more k hash functions to insert the same number of elements and achieve the minimum false positive. Thus, a smaller bloom filter implies less hash functions and a bigger filter implies the use of more hash functions to achieve the minimum false positive. In a next step, we analyze numerically the probability of false positive while varying the number of hash functions k and the number of inserted elements. Figure 2 depicts their effect on the false positive probability. We observe that for a given bloom filter with a fixed size, increasing the number of hash functions or the number of inserted elements increases the probability of false positives. Moreover, increasing the size of the bloom filter decreases the probability of false positives.

2) False positive rate: In the previous analysis, we have assumed that n elements are already inserted in the filter to compute the false positive probability. However, we are interested in the number of false positive when inserting a new element. Therefore, we start with an empty filter and we insert one element at each step. Let $p(n) = (1 - (1 - \frac{1}{m})^{nk})^k$ the probability of a false positive after inserting n elements with a fixed filter size m and k hash functions used per element. Let NB_{fp} the number of false positive items after one element is inserted in the filter. Initially $NB_{fp}(0) = 0$ where the filter is empty. Then we proceed as follows to compute the number of false positives per inserted elements. When a first element is inserted in the filter, we have $NB_{fp}(1) = NB_{fp}(0) + 1 \times p(0) + 0 \times (1 - p(0))$. By recurrence, we can computer the number of false positive after inserting n elements, which is equal to $NB_{fp}(n) = NB_{fp}(n-1) + 1 \times p(n-1) + 0 \times (1 - p(n-1))$. Then the false positive rate after inserting n elements can be written as follows.

$$fpr(n) = \frac{NB_{fp}(n)}{n} \tag{4}$$

This analysis tells us that bounding the number of false positives implies bounding the number of inserted elements in the Bloom Filter. Figure 3b depicts the effect of the number of inserted elements on the false positive rate. The circles marked plots present the false positive rate computed by the equation 4 and the square marked plots presents the estimation of the false positive rate according to the estimation $(\frac{1}{2})^k$ proposed in [4]. We observe that when the Bloom filter and the number of inserted elements are small, the estimation fits our formula. However, when the number of inserted elements is important or when the filter is large the lower bound of the false positive rate proposed in [4] is overestimated. In our work, we use multiple bloom filters from different users. Thus, we are interested in analysing the probability of a false positive in case of merging N Bloom filters or keeping them disjoint to query an element.

3) Merging or not users Bloom filters: Let N=$n_1 + n_2 + ... + n_s$ where s is the number of users sharing their bloom filters in our defense system and n_i is the number of elements inserted in each bloom filter. When merging different users Bloom filters, the probability of false positive is $fp_{merge}(N) = (1 - (1 - \frac{1}{m})^{Nk})^k$. In the second case, where we keep the filters disjoint, the probability to get a false positive is the probability of a false positive in at least one user's filter. Hence, we obtain: $fp_{disjoint} = 1 - (1 - p(n))^s$. Figure 4 depicts the difference between the two probabilities while varying the number of inserted elements and the number of hashing functions. The merged filter has a size of 8KB and each of the separated filter has a size of 4KB. We observe that merging the filters has a

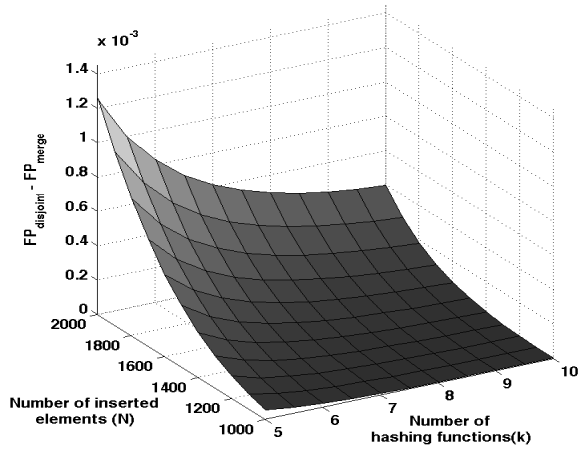

Fig. 4: The difference of probabilities of a false positive between bloom filters merging and keeping them disjoint before querying.

lower probability of a false positive than keeping all user's filters separated.

C. Analysis of hash based SMS filtering

An SMS filter is defined as a set of elements where each element represents one SMS. We have tried two hash functions to represents this set, where each SMS is hashed into k bits and stored into a list. We compared the performance of the two hashing algorithms MD5 and FNV [7] to create SMS hashes. In our experiment, we used 10^6 different SMS and we computed the number of false positive after inserting a new SMS hashed value in the list. Using a 16 bits hashed value of a SMS, empirical and theoretic results are close for both MD5 and FNV. The difference between empirical and theoretic results is around 0.01 for FNV and 0.05 for MD5. Therefore, our focus goes on the FNV based hashes which are dedicated to small messages like SMS. We found that FNV false positive probability is close to 10^{-9} for a list of size 4000 entries and a 40 bits hashed value. This probability becomes close to 0 with 400000 entries and 48 bits hashed values. Thus, the FNV algorithm provides a lower false positive rate and it was shown that its throughput is acceptable [8].

D. Simulation study of Bloom Filter and Hash functions

While the above experiments look into the theoretical performance of hashing and bloom filters to create SMS filtering lists, their performance depends on their implementation within the target host. In [9], authors state that a person receives on average one SPAM per day. We evaluated hashing functions and bloom filters for the different required filters of the Hinky platform: local, group and global. We have assessed their filtering capacity in terms of number of stored SMS, their storage cost and lookup time. We fixed the number of inserted elements for each filter as follows: 4000 elements for the local filter, 400000 elements for the group filter and 50000000 elements for the global filter. We implemented the local filter on an Android simulator and it is persisted in a SQLite database. The group and global filters are located on a host with an 2.4Ghz Bi-processor CPU machine and 4GB of memory. The two filters are persisted in a MySQL database.

Filter Type	Filtering Method	Parameters
Local	Hashing	FNV-40 bits
Group	Hashing	FNV-48 bits
Global	Bloom	k=30, m=256MB

TABLE I: Different filters representation in the Hinky platform.

1) Local filter: Using Bloom filters the filter capacity varies between 2000 and 4000 elements. This is due to the aging mechanism which relies on two sub filters as explained in section II-B1. However, the hashing set contains mostly the 4000 elements. Each element is a 40 bits FNV hashed value. When we need to insert a new SMS the last element is removed and replaced by the new one. The two sub Bloom filters have a 20KB size to achieve the 10^{-9} false positive rate. The hashing list requires the same storage with a 40 bits FNV hashed values. While inserting an element, we measured the time of its lookup and its insertion in the database if it does not exist. We found that by using the hashing method this time is around 0.3 seconds and 0.6 for a Bloom filter. The main reason for the difference is the cost to compute the k hash functions and access the k bits positions. Therefore, we observe that a hashing method is more performant than a Bloom filter for the local filter.

2) Group filter: For the same reason as above, the group filter capacity using the FNV hashing method is larger than a Bloom filter. That hashing list contains at most 400000 elements. The storage of these elements requires 2.06 MB using a Bloom filter and 2.3 MB using the FNV method with 48 bits hashed values. The false positive probability was fixed to 10^{-9} for the two methods. The lookup and insertion time for the hashing method is 2 times faster than the bloom filter.

3) Global filter: For the global filter with Bloom filter, we fixed the number of hashing functions K=30. The hashing based filter uses an MD5 hashing method with a 128 bits hashed values. The capacity of a Bloom filter varies between 0 and 5×10^7 inserted elements. We used a Cold Cache strategy for the filter where we remove all elements when the filter is full. A hashing based method provides the maximum filtering capacity of 5×10^7. To store the 5×10^7 elements in the filter with a false positive probability close to 10^{-9} the Bloom filter requires 256 MB and a hashing method based on MD5 128 bits hashed values requires 762 MB. We found that the lookup and insertion time for the Bloom filter is around 0.01 seconds. For the hashing method this time linear and becomes greater than 0.01 seconds after inserting 4000 elements

The table I shows the required filtering methods for each filter in our defense system.

IV. Implementation

We have implemented our defense system according to the architecture described in Figure 1. We have developed a communication protocol between the mobile and the server

components to register users, create filtering groups and to check an SMS. The different messages are the following:

- The *register* message: is used by the mobile user to register itself within the filtering service running on a dedicated server. The message carries the login and password provided by the user.
- The *join* message: is used by a mobile user to join a filtering group of its choice. The user needs to provide the name of the group which is the login of a trusted user providing its filter.
- The *leave* message: is used to leave a group by providing its credentials and the list of groups.
- The *reportSpam*: is used to report an SMS as a spam to the group filter. The message carries the user credentials and the hashed value of the SMS.
- The *isSpam*: is used to check if an incoming SMS is a spam or not. The collaborative server will check firstly the group filter, then the global filter. A mode where the group filter is pushed regularly to the mobile is available as well.

The different messages between the user and the collaborative server are carried over the HTTP/HTTPS protocols.

A. Mobile side implementation

On the mobile we implemented the local filter with the parameters identified in table I. The filter contains hashed values of SMS indicated as SPAM by the user. The mobile side application also provides a user interface to manage blocked SMS and the list of filtering groups. All the application data (filter, lists and groups) are stored in an SQLite data base.

B. Core Engine implementation

The collaborative server uses Jetty and MySQL to manage mobile clients requests to join groups, report spam as SMS and check SMS as spam or not. The server also maintains the global filter where an SMS identified as spam by many groups is inserted. The identification relies on a threshold function which computes the number of reports of a spam over existing groups. If the number exceeds a predefined value, the SMS hashed value is inserted in the global Bloom filter. To manage the global filter persistence, we use an approach close to the Prevayler project [10]. The Bloom filter is stored in a file which is loaded in memory as an object. The transactions are executed in memory over this object. Periodically, a snapshot is taken on the object to synchronize it to the system file.

C. A friendship mechanism for trust filtering

Users of the Hinky platform share their filters with the basic concept of *friend*. Each user can benefit from a local filter of another user by creating a friendship link with him. The link is created by sending a join message with the identifier of this friend. Therefore, when the mobile side application of Hinky checks if an SMS is a spam or not, it sends an *isSpam* request to the server. The server identifies the list of friendship links of the user. Then it does a lookup over their filters to identify if one of them contains the SMS to be checked. Therefore, each group is like a dynamic view over the set of users filters available on the server. When a user leaves a group (a friend), we only remove the identifier of his friend from the list of his friendship links. This mechanism has the advantage to be simple and it is driven by the user. It prevents from false reports since we only verify an SMS on trusted friends. A malicious user can not pollute a group filter since it need to be a trusted friend. We have to note that we used a direct friendship relation in our system. Each filter group viewed by a user contains only filters from his direct friends.

V. Conclusions

We have presented a solution to defeat SMS spam over smartphones using a user centric approach. Where users share their filters using a cooperative friendship mechanism. We believe that fighting SMS spam is a user problem rather than a telephony operator problem. The reason is that spammers are usually customers of the operator and they pay to get the SMS delivered to its destination. Therefore, operators have incentives, sometimes even contractual ones to provide the SMS service to all their customers including spammers. In our solution, we analyze two different methods to construct an efficient SMS filtering mechanism: Bloom filter and content hashing based lists. Our analysis targeted the rate of false positive which is an important metric due to the criticality of an SMS. We found that a hashing based approach provides better lookup time than a bloom filter and it supports the deletion operation which is missing in a Bloom filter. Therefore, a hashing list has a better capacity than a Bloom filter for this particular function. Currently, filtering users group's are created manually using a join mechanism. We are working on the automatic creation of these groups using social or business networks. We also plan to evaluate the energy consumption of our filtering approach on a smartphone.

References

[1] G. V. Cormack, "Email spam filtering: A systematic review," *Found. Trends Inf. Retr.*, vol. 1, no. 4, pp. 335–455, 2007.

[2] K. Li, Z. Zhong, and L. Ramaswamy, "Privacy-aware collaborative spam filtering," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 5, pp. 725–739, 2009.

[3] J. Yan and P. L. Cho, "Enhancing collaborative spam detection with bloom filters," in *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 414–428.

[4] A. Broder, M. Mitzenmacher, and A. Broder, "Network applications of bloom filters: A survey," in *Internet Mathematics*, vol. 1, 2005, pp. 636–646.

[5] F. Chang, F. Chang, and W. chang Feng, "Approximate caches for packet classification," in *In IEEE INFOCOM*, 2004, pp. 2196–2207.

[6] M. Yoon, "Aging bloom filter with two active buffers for dynamic sets," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, pp. 134–138, 2010.

[7] L. Noll, "Fowler/noll/vo (fnv) hash," in *http://isthe.com/chongo/tech/comp/fnv/*, 2004.

[8] O. Erdogan and P. Cao, "Hash-av: fast virus signature scanning by cache-resident filters," *Int. J. Secur. Netw.*, vol. 2, no. 1/2, pp. 50–59, 2007.

[9] cloudmark.com, "Taxonomy of current and potential mobile threats," in *http://www.cloudmark.com/releases/docs/wp_taxonomy_of_mobile_threats.pdf*, 2007.

[10] Prevayler, "Objet persistence library for java," in *http://www.prevayler.org*, 2006.