Enhancing Collaborative Spam Detection with Bloom Filters

Jeff Yan Pook Leong Cho
Newcastle University, School of Computing Science, UK
{jeff.yan, p.l.cho}@ncl.ac.uk

Abstract

Signature-based collaborative spam detection (SCSD) systems provide a promising solution addressing many problems facing statistical spam filters, the most widely adopted technology for detecting junk emails. In particular, some SCSD systems can identify previously unseen spam messages as such, although intuitively this would appear to be impossible. However, the SCSD approach usually relies on huge databases of email signatures, demanding lots of resource in signature lookup, storage, transmission and merging. In this paper, we report our enhancements to two representative SCSD systems. In our enhancements, signature lookups can be performed in constant time, independent of the number of signatures in the database. Space-efficient representation can significantly reduce signature database size. A simple but fast algorithm for merging different signature databases is also supported. We use the Bloom filter technique and a novel variant of this technique to achieve all this.

1. Introduction

Spam (junk bulk email) is an ever-increasing problem. It causes annoyance to individual email users but also imposes significant costs on many organisations. To date, statistical spam filters are probably the most heavily studied and the most widely adopted technology for detecting junk emails. However, among other disadvantages, these filters need to be regularly trained, particularly when the filters result in excessive numbers of "false positive" or "false negative" decisions. In particular, such systems fail to detect spam that cannot be predicted by the machine learning algorithms on which they are based. Such filters also cannot identify spam that is sent as an image attachment to an otherwise unobjectionable email message. In addition, as content-based filters, they are language-dependent (e.g. a filter trained for English is useless in detecting spam in Chinese, and vice versa) and vulnerable to various content-manipulation attacks (e.g. so-called "filter poisoning").

Signature-based Collaborative Spam Detection (SCSD) systems, e.g. Razor [7] and Distributed Checksum Clearinghouse (DCC) [3], are an attractive complement to statistical spam filters. As an alternative approach, these systems provide a promising solution addressing all the above problems facing statistical filters. In particular, systems like DCC can identify previously unseen spam messages as such, although intuitively this would appear to be impossible.

However, SCSD systems usually rely on huge databases of email signatures, demanding expensive computers and lots of resource in signature lookup, storage, transmission (over the Internet) and merging. For example, a busy Razor or DCC server usually uses a dedicated computer. A dedicated Razor server typically handles up to 200 million queries per day. The number of active signatures it maintains is about 10 million at any time, and the database size exceeds 320 MB [8]. A dedicated DCC server typically handles up to 10 million requests per day. Its database is typically of about 1 GB (up to 5 GB), storing more than 20 million signatures [11].

We have performed an analysis of DCC source code and confirmed that a standard technique of hash table with internal chaining (dealing with collisions) is used to support all signature insertion, lookup and deletion operations. A large collection of message signatures, a hash table used for organising these signatures, and other information are all stored, leading to a huge database as well as expensive computation. Techniques used in Razor are not publicly known. The source code of Razor's server program is not publicly available, either. However, the size of its signature database, together with technical information available on the Internet (e.g. the largest message signature used in Razor is of 20 bytes [7]), suggests that at least signature storage, transmission and merging could be optimised in Razor.

In this paper, we propose some enhancements to both Razor and DCC. In our enhancements, signature lookups can be performed in $O(1)$, i.e. constant time, independent of the number of signatures in the database. Space-efficient representation can significantly reduce signature database size (e.g. by a factor of 16 or more for the Razor system), before any data compression algorithm is applied. This also implies much less traffic when signature databases are exchanged over the Internet. A simple but efficient algorithm for merging different signature databases is also supported.

We have achieved all this using the Bloom filter technique [1] and a novel variant of this technique. Our variant extends the standard Bloom filter scheme to support counting, heuristics for reducing counting errors, and an innovation for storage saving. This variant can also be applied to other distributed applications.

The rest of this paper is organised as follows. Section 2 provides technical backgrounds of this paper by briefly reviewing both the signature-based collaborative spam detection approach and the Bloom filter technique. Section 3 discusses enhancing the Razor system with Bloom filters. Section 4 introduces our new Bloom filter variant, and discusses enhancements it can introduce to the DCC system. Section 5 reports a simulation study, which shows the performance improvement the new variant can achieve in reducing counting errors. Section 6 concludes with a summary of the main contributions of this paper and a brief discussion of ongoing and future work.

2. Technical background

2.1. Signature-based collaborative spam detection

The signature-based collaborative spam detection approach is based on simple but powerful insights. For example, Razor implements the following idea: if a message has been identified elsewhere as spam by somebody trustworthy, then this human effort shall be shared/reused.

DCC also relies on a simple but insightful observation: spam by definition is unsolicited *bulk* email, so we can detect spam by checking for "bulkiness". That is, when a message that has been seen many times elsewhere on the Internet reaches you, if it is not from any person, organisation or email list that appears on your so-called "white list", then it is safe for the email system to treat it as spam and discard it. This is a clever way of detecting and dealing with spam email messages (including those unforeseeable new ones) without checking the message content.

As indicated, both DCC and Razor are **signature**-based. In the simplest conceptual form of both systems, one *signature* (i.e., *digest* or *checksum*) is computed with a cryptographic hash $h()$ to represent each message. Since any slight change in input to such a hash will dramatically change its output, $msg1$ and $msg2$ are considered the same if and only if $h(msg1) = h(msg2)$. The use of a crypto hash also addresses users' privacy concern, since anybody who receives a message signature will not be able to reverse engineer it to get the message text.

In the actual implementations of these two systems, multiple different signatures are calculated for the same message in some scenarios. To simplify our discussion, unless otherwise indicated, we assume in this paper that an email message is represented by a single signature. However, our discussions can be generalised to the actual case easily.

A simple illustration of how DCC works is as follows. A DCC server collects and accumulates counts of signatures for email messages. To decide whether a new message is spam, a DCC client queries the server using a signature of the message. If the count number for the signature returned by the server is larger than a local threshold value set by the user, then the message is marked as spam.

In Razor, a server maintains a database of signatures for identified spam. That is, an end-user identifies a spam message and then reports its signature to the server serving her. Other users will query the server to detect spam in their mail boxes: if a particular email message already has its signature appearing in the server database, then it is identified as being spam.

Both DCC and Razor are **collaborative** in nature. Both systems run a distributed network of (signature) servers, each serving a particular part of the user population and collecting signatures from that particular community. Signature databases are periodically synchronised among all servers. In this way, each user's effort can be reused by many others.

The following issues are critical to the success of SCSD systems:

- **Near-replica identification.** Near-replicas are similar messages with minor differences. Spammers often use them to evade detection. Since any slight change in an input to a crypto hash function will dramatically change its output, it seems impossible to correlate near-replica messages by examining their signatures computed with such a hash. It would be very useful to create a "fuzzy" hash function that will produce similar hashed values for similar inputs. This hash should also be robust against a number of attacks such as random addition, dictionary substitution and perceptive substitution (e.g. substituting "Viagra" by "V1agr@").

- **Trust.** Spammers can cheat so as to defeat any spam detection system. How would you differentiate trustworthy users and spammers in the same community so that their updates to the servers are treated differently? A proper reputation system is essential, in particular for Razor and the like systems.

Solutions adopted by Razor and DCC to address these issues appear to be effective and sophisticated enough to be deployed on a large scale, as the accuracy and popularity (see [7, 3]) of both systems suggests. However, much improvement can be done, as suggested by our preliminary analysis, but they are beyond the scope of this paper.

2.2. Bloom filters

Conceived by Burton H Bloom in 1970, a Bloom filter is a space-efficient data structure supporting fast membership testing [1]. It is a bit vector B of m bits, each set to zero initially. To insert an element x into B, first compute $h_1(x), ..., h_k(x)$ with k independent, random hash functions, each mapping x into the range $\{0, 1, ..., m-1\}$; then, set $B[h_1(x)] = ... = B[h_k(x)] = 1$. To query if y is a member in the filter, $h_1(y), ..., h_k(y)$ are computed. If $B[h_1(y)] = ... = B[h_k(y)] = 1$, then answer Yes, else answer No.

A Bloom filter does not introduce false negatives (answering No when an element is actually in the filter), but it can cause small false positives (answering Yes when querying an element that is not in the filter). A false positive occurs when an element y is not stored in the filter, but accidentally (by coincidence) $B[h_1(y)], ..., B[h_k(y)]$ are all set to 1. The probability that a false positive occurs, or the false positive rate, for a Bloom filter can be made as small as desired, and it can be calculated as follows.

The probability that one hash fails to set a given bit is $1-1/m$. After n elements are inserted into the Bloom filter, the probability that a specific bit is still 0 is: $(1-1/m)^{kn}$. The probability of a false positive, f, is the probability that a specific set of k bits are 1, and it can be estimated with the following approximation:

$$f \approx (1-(1-1/m)^{kn})^k \approx (1-e^{-kn/m})^k \qquad (1)$$

Three performance metrics in a Bloom filter can be traded off: k (computation time), m (storage size) and f (false positive). The false positive rate f is minimised when $k = ln2 \times m/n$, and $f_{min} = (\frac{1}{2})^k \approx (0.6185)^{m/n}$. As m grows in proportion to n, f will decrease.

It is worthwhile to note that the claim by Bloom in his original analysis [1] that the false-positive rate $f = (1-(1-1/m)^{kn})^k$ is incorrect. He implicitly assumed that the event "$B[h_1(y)] = 1$", the event "$B[h_2(y)] = 1$", ..., and the event "$B[h_k(y)] = 1$" are independent. However, this assumption is not necessarily true. For example, that $B[h_1(y)]$ is set to 1 can have an impact on the outcome of $B[h_k(y)]$. Nonetheless, the false positive rate of Bloom filters observed in simulations matched well with its theoretical estimation given by Equation (1), as shown in empirical studies such as [9].

Notable applications of Bloom filters in computer security include the following. In early 1990's, Spafford [12] proposed to use a Bloom filter to build a proactive password checker that could quickly tell whether a password candidate was in a collection of weak passwords. Recently, a new Bloom filter variant was introduced to store portions of network packets for the purposes of payload attribution in forensics [10]. A brief survey of application of the Bloom filters in other contexts is also included in [10].

3. Enhancing the Razor system with Bloom filters

Intuitively, if signature databases are organised with Bloom filters in the Razor system, we could achieve fast signature lookups, significantly reduce the database size, and obtain an efficient algorithm for merging signature databases. However, the following two problems have to be addressed before applying the Bloom filter technique to Razor.

- *Choosing proper hash functions*. A popular way of constructing Bloom filters is to use MD5 or other cryptographic hash functions, as described in [4]. However, such a construction and the like do not work well in our setting, as will be discussed below.
- *Signature revocation*. Occasionally, a Razor server has to revoke from its database signatures that are falsely identified as spam. However, a Bloom filter does not support deletion: to set a bit to zero could delete too many elements!

Choosing proper hash functions. Fan et al [4] used MD5, a message digest function that hashes arbitrary length strings to 128 bits, to implement their Bloom filter. They chose $k = 4$, and the k hash functions were constructed as follows: for each x to be inserted into the filter, they first applied the MD5 to get a 128-bit hashed value of x. The hashed value was then divided into four 32-bit words. Taking the modulus of each 32-bit word by m, the size of the bit vector, gave an index in the vector.

It would appear to be straightforward to generalise the above method to construct the Bloom filters with arbitrary k hash functions as follows:

$$h_i(x) = (\text{the i-th chunk of MD5}(x)) \bmod m,$$

where $i = 1, ..., k$ and $k|128$ (i.e., 128 is dividable by k). However, the actual number of bits that can be utilised in the Bloom filter will be bounded by $min(m, 2^{128/k})$. See Fig.1(a), which shows a scenario where all bits in the filter are reachable and thus can be utilised, and Fig.1(b), which shows a scenario where the number of utilisable bits are smaller than the filter size m. Therefore, the above construction does not leave much room for the choice of k. For example, when $k = 8$, the number of utilisable bits in the filter is $min(m, 2^{16})$, which is too small for most applications! It is also meaningless to trade off other parameters by increasing m in this kind of scenario.

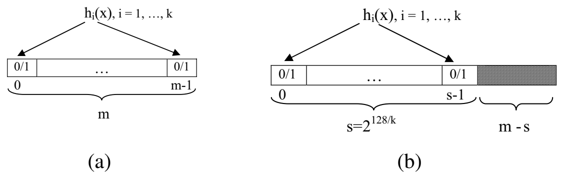

Figure 1. (a) A scenario where all the bits in the Bloom filter are utilisable – no unreachable bits. This is the case when $2^{128/k} \geq m$ (assuming that MD5 is used). A good example is the Bloom filter built by Fan et al [4], where $k = 4$. (b) A scenario where only a part of the Bloom filter can be utilised. This is the case when $2^{128/k} < m$ (also assuming that MD5 is used). The unreachable bits, of size $m - 2^{128/k}$, are shaded.

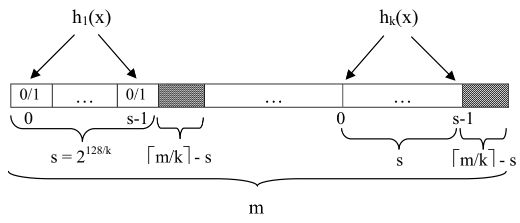

Figure 2. Each h_i hash maps x into the i-th chunk of the Bloom filter. The shaded part highlights unreachable bits in each chunk of the filter. Assume that MD5 is used.

In Razor, each spam signature is typically a 160-bit hashed value calculated with SHA-1. Suppose that the following k hash functions are used to construct the Bloom filters for Razor:

$$h_i(x) = (\text{the } i\text{-th chunk of SHA-1}(x)) \bmod m,$$

where $i = 1, ..., k$ and $k|160$. The number of utilisable bits in the filter can be increased, but bounded by $min(m, 2^{160/k})$. The same difficulty still exists. For example, when $k = 8$, the number of utilisable bits in the filter is determined by $min(m, 2^{20})$, which is not large enough for many applications.

That is, although Bloom filters as constructed in [4] empirically achieved good performance, they are not a good choice in Razor. Such constructions cannot be easily generalised, either.

One partial solution is to divide the Bloom filter into k chunks, and each h_i hash maps x into the i-th chunk of the filter (when necessary, a modulus of the hashed value $h_i(x)$ by $\lceil m/k \rceil$ should be taken). This can increase the utilisable bits in the filter by a factor of k. But when m is sufficiently large, in each chunk of the filter, $\lceil m/k \rceil - 2^{160/k}$ (if SHA-1 is used) or $\lceil m/k \rceil - 2^{128/k}$ (if MD5 is used) bits will never be reachable. That is, in total, $m - k * 2^{160/k}$ or $m - k * 2^{128/k}$ bits are still unreachable in the filter (see Fig.2). To address this, a new hash function $\hbar_i$ could be introduced to map the collection of $h_i(x)$ values into the range $\{0, 1, ..., \lceil m/k \rceil - 1\}$.

In future, Razor might want to use a longer hash value representing a spam message, avoiding the above inconvenience all together in the first place. But for now, we can also address this problem by constructing Bloom filters in a different way. For example, universal hashing [2] is a good alternative building block, as shown in [9]; it is also applicable in our setting.

The class of universal hash functions is of the following form:

$$h_{c,d}(x) = ((cx + d) \bmod p) \bmod m \qquad (2)$$

where p is a prime, m, c, d are integers, $0 < c < p$ and $0 \leq d < p$. Such hash functions map a given universe U of keys into the range $\{0, 1, ..., m-1\}$. Construction of k such hash functions will be discussed in Section 5.

Signature revocation. Although a Bloom filter does not support deletion, there is a simple solution to support signature revocation in the Razor system. We can build a Bloom filter for all spam signatures, which we call the Spam Bloom filter (SBF), and build another for all revoked signatures, which we call the Revocation Bloom filter (RBF). Thus, spam detection becomes membership testing in these two Bloom filters. For example, we can first look up the SBF, and then the RBF. The results will be decided as follows.

1. If x is not in SBF, it is not spam;
2. If x is both in SBF and in RBF, it is not spam; if x is in SBF but not in RBF, it is spam.

The order for membership testing can be turned around. Which order is better (more efficient) really depends on the situation. If you expect more legitimate messages than spam, probably you should look up the SBF first.

Other concerns and results in applying the Bloom filters to Razor are now straightforward, as follows:

- **Signature lookup**. Lookups in SBF and RBF are both $O(1)$.
- **Storage saving**. With a Bloom filter, only m bits are required to record n distinct signatures, each of 160 bits. However, when such a collection of signatures was not organised with a Bloom filter, its actual storage would be: $160 * n +$ the size of indexing hash tables.

Table 1 shows storage saving achieved by a Bloom filter under different (m, n, k) configurations. To simplify the calculation, the storage compression rate (CR) was estimated with the formula CR $= 160 * n/m$.

m/n	k	CR	False positives
16	4	10	2.394×10^{-3}
10	8	16	8.455×10^{-3}
16	8	10	5.745×10^{-4}
40	8	4	1.166×10^{-6}
40	16	4	1.948×10^{-8}

Table 1. Storage saving and trade-offs between $m/n, k, f$ in a Bloom filter

- **Signature database merging**. When each Razor server uses the same parameters (m, n, k) and the same hash functions to build their Bloom filters, there is a simple but fast algorithm for merging signature databases: only a Bloom filter is needed to be transmitted from one server to another, and merging multiple databases is simply to OR the Bloom filters bit by bit.
- **False positives and negatives**. In this enhancement to the Razor system, we cannot eliminate false positives in spam detection. The Revocation Bloom Filter is also likely to introduce some false negatives. But both false positives and negatives can be tuned to be very small. For example, Table 1 shows the false positive rates in a Bloom filter under a number of different configurations. The figures are estimated with Equation (1).

  On the other hand, false positives and negatives occur in Razor even before any change is introduced to the system. For example, false positives will occur when a legitimate message is reported as spam and its signature included in a spam database. False negatives will occur when a database is not updated to include new spam signatures in time. However, Bloom filters do not provide a solution to address these false positives and negatives.
- **Signature aging**. It is difficult to purge individual signatures from a Bloom filter. However, one simple way of supporting signature expiration is to organise a signature database with a number of temporally ordered Bloom filters, rather than a single one. Signatures reported in the same period will be inserted into the same filter. When all signatures in a filter has become inactive for a predefined period, the filter will be discarded.

In our view, even if there is a small additional chance of false positives/negatives introduced by Bloom filters, it will be outweighed by the advantages they introduce.

4. Enhancing the DCC system with Bloom filters

The DCC system requires to keep track of the number of times a message has been reported to a server, i.e. the occurrence count of the message. A Bloom filter cannot record occurrence counts, but an intuitive extension to the standard scheme can support counting as follows. The extended Bloom filter is an array c of m cells, each being set to zero initially. Each cell works as a counter. When an element x is inserted or deleted, the counts $c[h_1(x)], ..., c[h_k(x)]$ will be incremented or decremented accordingly.

This extension was first reported by Fan et al [4]. However, they did not look into some useful details. For example, it was not discussed how to tell how many times an element x had been inserted into the filter, probably because this was not relevant in their application. The answer is simple: $\min(c[h_1(x)], ..., c[h_k(x)])$ tells the number of occurrences of x witnessed by the filter, although this figure occasionally might be just an approximation that is larger than the real occurrence count. Another useful detail missing in [4] will be discussed later on.

When such an intuitive extension is applied to the DCC system, the following features can be achieved.

- **Signature lookup** is still done in $O(1)$, independent of the number of signatures.
- **Signature deletion** is supported by this extended Bloom filter. But unlike in Razor, signature revocation in DCC is not essential for the purpose of spam detection. A DCC server only accumulates the occurrence count of each reported message, and it does not care whether a particular message is spam or not. However, it is still useful (e.g. for reducing the size of a signature database) to apply this signature deletion technique to purge identified useless signatures individually.

  Since our discussions about counting, i.e. the insertion operation, in this Bloom filter extension can be easily extended to the deletion operation, unless otherwise stated, we do not discuss the deletion case in the rest of this paper.
- **Signature storage**. Only $m * sizeof(cell)$ bits are required to store n signatures and their occurrence counts. The DCC end-users often use a threshold value $t = 20$ to determine whether a message is spam or not. That is, a message that has been seen for 20 times somewhere else will be considered as spam, if it is not from someone appearing on your white list. Therefore, 5 bits per cell in the filter might be sufficient in the DCC. Moreover, to provide more flexibility (e.g. some users might want to use a threshold value larger than

20), we can allow each cell in the filter to reach 31. If a count ever exceeds 31, we can simply let it stay at 31.

All this indicates significant improvement over the current solution in DCC, which has to store the following all together:

- n signatures, occupying $n * sizeof(signature)$ bytes;
- n occurrence counts, of $n * sizeof(count)$ bytes, and
- a huge hash table created for such a collection of signatures.

In addition, counts larger than 31 can easily be supported in each cell of the filter, which implies more storage consumption though.

- **False positives**. A false positive occurs when an element x does not occur so often, but accidentally $\min(c[h_1(x)], ..., c[h_k(x)])$ is larger than its actual number of insertions[1]. The probability of such false positives, denoted by f_I, is the false positive rate (or counting error rate) in this intuitive Bloom filter extension. We will resort to simulations for an analysis of f_I.

  However, such a false positive does not necessarily lead to a false identification of a legitimate email as spam in DCC. A false positive in DCC occurs only when a particular email message has its occurrence count reaching or exceeding the threshold value t (i.e. $c[h_1(x)], ..., c[h_k(x)]$ all reach t at least), although it in fact has not occurred so often. Since the probability that a counter is increased j times is a binomial random variable:

$$P(c_i = j) = \binom{nk}{j}(1/m)^j(1-1/m)^{nk-j}, \quad (3)$$

  the false positive rate in DCC, f_{dcc}, can be estimated for a given t by

$$f_{dcc} \approx \Big(\sum_{j=t}^{\max j}\binom{nk}{j}(1/m)^j(1-1/m)^{nk-j}\Big)^k, \quad (4)$$

  where $\max j$ is the maximal number of times the message x has occurred. Intuitively, $f_{dcc} < f_I$.

- **Signature database merging**. We assume that each end-user reports email messages she has received to no more than one DCC server. This is a realistic assumption, since each DCC server is usually designated to serve a particular part of the user population. Thus, when each DCC server uses the same parameters (m, n, k) and the same hash functions, a simple but fast algorithm for merging signature database can be supported: cell by cell addition.

  For the first round of database synchronisation, an extended Bloom filter is needed to be transmitted from one server to another. However, for any subsequent round of synchronisation, we can further reduce traffic exchanged between DCC servers by transmitting a delta Bloom filter only. For example, the first synchronisation between servers a and b at time interval t_0 may require each server to transmit its own Bloom filter to the other, and then the merging can be done as follows. For $i = 0, ..., m-1$,

$$c^{t_0}_{sync}[i] = \begin{cases} 31 & \text{if } c^{t_0}_a[i] + c^{t_0}_b[i] > 31 \\ c^{t_0}_a[i] + c^{t_0}_b[i] & \text{otherwise} \end{cases}$$

  But for the next subsequent synchronisation between the two servers at time interval t_1, server a just need transmit the following delta Bloom filter to server b:

$$c^{t_1}_{\Delta_a}[i] = c^{t_1}_a[i] - c^{t_0}_a[i], \ i = 0, ..., m-1,$$

  and server b just need transmit its own delta Bloom filter to server a:

$$c^{t_1}_{\Delta_b}[i] = c^{t_1}_b[i] - c^{t_0}_b[i], \ i = 0, ..., m-1.$$

- **Signature aging**. The scheme sketched for Razor in the previous section is also applicable here.

As such, it appears that this intuitive Bloom filter variant is well suitable for DCC. However, the following counterexample suggests that further refinements are needed.

Assume that each DCC server constructs its (extended) Bloom filters in exactly the same way: the same hashes and the same parameters (k, m, n) are used. Also assume that $k = 3$, without loss of generality. Also assume that in server a's Bloom filter, we have

$$c[h_1(x)] = 2, \ c[h_2(x)] = 5, \ c[h_3(x)] = 8.$$

Thus, $Count(x) = 2$. That is, message x has been reported to this server at most twice. Similarly, in server b's Bloom filter, we have

$$c[h_1(x)] = 4, \ c[h_2(x)] = 4, \ c[h_3(x)] = 3.$$

Thus, $Count(x) = 3$. That is, message x has been reported to this server at most three times. We should have $Count(x) = 5$ when two servers have completed synchronising their signature databases. However, the merging algorithm will give $Count(x) = min(6, 9, 11) = 6$!

1 This definition considers only insertions, having ignored the case of deletion.

The lesson is that many counters in the filter, before and after merging, could increase rapidly because of *coincidental hits*, by which a single cell is used by two or more elements.

We introduce a *refined* extension of the Bloom filter to address the above problem. In this extension, All else remain as in the intuitive extension, except that the following heuristic (which we refer to $\mathbf{H_1}$) will apply: **when x is inserted into Array c, among counts $c[h_1(x)], c[h_2(x)], ..., c[h_k(x)]$, only those that equal to $min(c[h_1(x)], ..., c[h_k(x)])$ will be increased by one.**

Return to the above scenario, and suppose that each server witnesses one more x. Then, with the intuitive extension, Server a has

$$c[h_1(x)] = 3,\ c[h_2(x)] = 6,\ c[h_3(x)] = 9,$$

and $Count(x) = 3$; Server b has:

$$c[h_1(x)] = 5,\ c[h_2(x)] = 5,\ c[h_3(x)] = 4,$$

and $Count(x) = 4$. However, after database merging, the system will have:

$$Count(x) = min(8, 11, 13) = 8.$$

On the contrary, with the refined extension, we will have

$$c[h_1(x)] = 3,\ c[h_2(x)] = 5,\ c[h_3(x)] = 8,$$

and $Count(x) = 3$ in Server a, and

$$c[h_1(x)] = 4,\ c[h_2(x)] = 4,\ c[h_3(x)] = 4,$$

and $Count(x) = 4$ in Server b. After merging, we will have an accurate result:

$$Count(x) = min(7, 9, 12) = 7.$$

That is, the counters in the refined extension do not increase as rapidly as in the intuitive extension, both before and after merging!

Table 2 compares the counter growth in these two extended Bloom filter schemes in Server b. It shows clearly that the refined extension has a better performance in controlling undesirable counter incrementing.

It is worthwhile to note that a correct implementation of the refined Bloom filter extension implies an additional heuristic $\mathbf{H_2}$: **when x is inserted, if any two or more of $h_1(x), h_2(x), ..., h_k(x)$ hit the same counter, then that counter should be increased only once.** If we say that $\mathbf{H_1}$ is introduced to address *global* coincidental hits caused by multiple elements, then the heuristic $\mathbf{H_2}$ addresses *local* coincidental hits caused by a single element. $\mathbf{H_2}$ should also be implemented in the intuitive Bloom filter extension in order to reduce false positives – this is another insight that was missing in [4].

Occurence	Intuitive extension			Refined extension		
of x	$c[h_1(x)]$	$c[h_2(x)]$	$c[h_3(x)]$	$c[h_1(x)]$	$c[h_2(x)]$	$c[h_3(x)]$
4	5	5	4	4	4	4
3	4	4	3	3	3	3
2	3	3	2	2	2	2
1	2	2	1	1	1	1
0	1	1	0	1	1	0

Table 2. Counter growth in two extended Bloom filter schemes: an example (as seen by Server b)

We define the false positive rate in the refined extension, f_R, the same as f_I is defined. f_R cannot be subjected to mathematical analysis. However, intuitively, $f_R < f_I$. In addition to the reduced false positives, the refined Bloom filter extension enjoys all other nice features in the intuitive extension.

Another innovation we introduce to our Bloom filter extension is to reduce its storage cost by splitting it to two parts: a base filter and a number of hash tables. We rely on a simple intuition: it is a waste of space to allocate each counter the number of bits large enough to accommodate the largest count that will be recorded, if the filter is expected to maintain many small counts and the discrepancy between the sizes of the small and large counts is large.

Instead, we introduce a base filter that has a uniform cell size of $s + 1$ bits, assuming that the expected largest small count value is not larger than 2^s. A one-bit flag in each cell indicates whether this count has additional bits, which, if any, are stored somewhere else. That is, a large count has only part of bits (e.g. its lower half) kept in the base filter. Its other part could be stored in a hash table, indexed by the offset of the count in the base filter. To reduce the space occupied by this hash table, where each index requires $\lceil \log_2 m \rceil$ bits, we virtually divide the base filter into a number of, say N, chunks. Then, instead of having a large hash table for the whole filter, we organise N small hash tables, each with the index size reduced to $\lceil \log_2(m/N) \rceil$, and each storing additional bits of large counts in a corresponding chunk only. Preliminary results suggest that this technique is promising. The details will be reported in a forthcoming paper.

5. A simulation study

It would be very interesting to evaluate how much more the refined Bloom filter extension would improve f_{dcc} than the intuitive one did, using empirical data collected from various DCC servers. However, such data collection has proved to be difficult. The DCC developer did not like the idea that an additional DCC server gets connected to the whole DCC network "for temporary, purely academic purposes" [11].

Instead, we have run a series of simulations to compare the false positive rate that could occur in both Bloom filter extensions, i.e. f_I and f_R. Additional reasons supporting such a decision are as follows.

First of all, both extended Bloom filters are data structures of general interest. For example, both can be applied to applications where it is relevant to support fast membership testing and distributed counting[2]. Therefore, we do not limit our simulation design to the purpose of enhancing DCC only, but also aim to gain a good understanding of how both extensions (the refined extension in particular) will perform in a general setting. To the best of our knowledge, no such effort has been reported in the literature.

Second, f_{dcc} is smaller than the false positive rate of the specific Bloom filter extension implemented in DCC. That is, $f_{dcc} < f_I$ or $f_{dcc} < f_R$. Therefore, f_I or f_R observed in our experiments can be used as an upper bound of f_{dcc}.

5.1. Simulation design

Without loss of generality, universal hashing is used to construct both extended Bloom filters in our simulations. One advantage of this construction is its simplicity and efficiency. Following the practice in [9], we use $p = 2,100,000,011$, and we generate $2k$ pseudo-random numbers, each pair being used as c and d to define a hash in the form of Equation (2)[3].

We use 10,000 distinct keys (i.e., elements to be inserted to the Bloom filters) in our simulations. They are integers randomly drawn from a universe $A, A = \{1, 2, ..., p-1\}$. The k hash functions are applied to each of the keys and the corresponding cells in the Bloom filters are incremented accordingly. In the intuitive extension, cells $c[h_1(x)], ..., c[h_k(x)]$ will all be incremented when x is inserted into the filter, and the heuristic $\mathbf{H_2}$ will also be enforced. In the refined extension, both $\mathbf{H_1}$ and $\mathbf{H_2}$ are enforced, and thus only the cell with a value equalling to $min(c[h_1(x)], ..., c[h_k(x)])$ will be incremented (by one only).

Our experiments are designed as follows.

Experiment 1. Each of the 10,000 elements is inserted sequentially into the filter. This entire process is repeated 20 times. The whole insertion sequence is as follows.

$$\underbrace{x_1, x_2, ..., x_{10,000}}_{Round_1}, \;......, \underbrace{x_1, x_2, ..., x_{10,000}}_{Round_{20}}$$

Experiment 2. Each element is inserted 20 times repeatedly into the filter. The entire process continues until all the 10,000 elements have been inserted. The whole insertion sequence is as follows.

$$\underbrace{x_1, ..., x_1}_{20}, \underbrace{x_2, ..., x_2}_{20}, \;... ...,\; \underbrace{x_{10,000}, ..., x_{10,000}}_{20}$$

Experiment 3. Each of the 10,000 elements is inserted into the filters 20 times, but the sequence for insertion is random. We apply the classical Fisher-Yates shuffle algorithm [6], converting the insertion sequence in Experiment 2 into a random sequence. Each element in the random sequence is then inserted sequentially into the filter.

Experiment 4. Each of the 10,000 elements is inserted into the filters in a random order, and each inserted a random c times ($c \in [0, 20]$). For each element x_i, we generate an integer c_i, uniformly distributed on the range $[0, 20]$. Then, we organise all the elements in the following sequence.

$$\underbrace{x_1, ..., x_1}_{c_1}, \underbrace{x_2, ..., x_2}_{c_2}, \;... ...,\; \underbrace{x_{10,000}, ..., x_{10,000}}_{c_{10,000}}$$

Some elements may not appear in the sequence since c_i can be zero. Let us assume there are l non-zero elements in the sequence. We apply the Fisher-Yates algorithm to shuffle the l elements into a random sequence, and then sequentially insert each element into the filter.

Experiment 5. The unshuffled sequence in Experiment 4 is inserted into the filter. That is, all the elements are sequentially inserted into the filter, and each element inserted repeatedly a random c_i time (c_i is uniformly distributed on the range [0,20]).

Experiment 6. This experiment is the same as Experiment 4 except that c_i, the number of insertions for each element, is modelled as a Poisson random variable with parameter $\lambda = 10$.

Experiment 7. Same as Experiment 4 except that c_i is modelled as a Poisson random variable with $\lambda = 20$.

Experiment 8. Same as Experiment 4 except that c_i is uniformly distributed on the range $[0, 40]$.

In each of the above experiments, different (m, n, k) configurations are tested for both extensions. For each configuration, the simulation is repeated for 1,000 different sets of hash functions, i.e. 1,000 rounds. The mean and the standard deviation of the false positive rate will be noted for each configuration for both extensions.

False positives are obtained by implementing a FP counter for each extension, which is initialised to zero at

2 Some counts returned by both extended Bloom filters might not be accurate, due to coincidental hits. However, the number of such "approximate counts" in the refined extension can be very small, as shown in the later part of this paper.

3 It is worthwhile to note: k hash functions constructed this way are not necessarily independent, strictly speaking. However, when they were used to build Bloom filters, the empirical false positive rate of these filters met its theoretical expectation [9]. We have repeated the experiments introduced in [9] and confirmed this result.

In our future work, we will apply additional constraints as suggested by Knuth [5] to construct k independent, random (universal) hashes and then repeat experiments discussed in this section to see whether any new findings will be found.

the beginning of each round of simulation. After all the insertions are done in a round, all distinct elements that are inserted into the filter are identified. (That c_i can be zero in some experiments implies that some elements will not be inserted.) A list of such distinct elements is then run through to query the Bloom filter identifying those with an erroneous count in each extension. Whenever such elements are found, the FP counter will be incremented accordingly. For example, in Experiments 1-3, for an element x_i, if $min(c[h_1(x_i)], ..., c[h_k(x_i)]) \neq 20$, then it has an erroneous count and the FP counter increases by one. In Experiments 4–8, for an element x_i, if $min(c[h_1(x_i)], ..., c[h_k(x_i)]) \neq c_i$, then it has an erroneous count and the FP counter increases by c_i.

The false positive rate is calculated by $\frac{FP}{10,000}$ in each round of Experiments 1–3, and by $\frac{FP}{l}$ in each round of Experiments 4–8, respectively.

5.2. Simulation results and observations

Tables 3–10 show the results of each experiment, including the false positive rate of both extended Bloom filters under different configurations, and reduction in false positives achieved by the refined extension. A pictorial comparison of the false positive rates in both extensions in these experiments can be found in [13].

The main reasons that we choose to provide detailed experimental result data in this paper include the following. The false positive rates observed for both extensions in our experiments are useful empirical estimates. People can use these data without repeating the simulations by themselves. In particular, the only way of estimating the false positive rates in the refined extension is to resort to simulation, which is unfortunately very time-consuming.

Filter	$k = 4$		$k = 6$		$k = 8$	
size m	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.
80K	2.390E-2	1.556E-3	2.154E-2	1.485E-3	2.548E-2	1.559E-3
160K	2.372E-3	5.013E-4	9.446E-4	2.961E-4	5.686E-4	2.375E-4
320K	1.860E-4	1.381E-4	2.570E-5	5.089E-5	4.500E-6	2.073E-5

(a)

Filter	$k = 4$		$k = 6$		$k = 8$	
size m	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.
80K	5.840E-3	7.786E-4	4.167E-3	6.633E-4	4.316E-3	6.430E-4
160K	5.107E-4	2.323E-4	1.591E-4	1.250E-4	7.720E-5	8.637E-5
320K	3.450E-5	5.692E-5	3.100E-6	1.733E-5	3.000E-7	5.469E-6

(b)

Filter	Reduction Rate		
size m	$k = 4$	$k = 6$	$k = 8$
80K	4.094	5.170	5.903
160K	4.645	5.937	7.365
320K	5.391	8.290	15.000

(c)

Table 3. The false positive rates in $Experiment\,1$: (a) for the intuitive extension; (b) for the refined extension; (c) reduction rate achieved by the refined extension.

As observed from the experiments, **false positive rates in both extensions are controllable, and can be made very small by proper choice of m/n and k.** However, the refined extension has never yielded more false positives than the intuitive extension, given the same configuration. Instead, the former can effectively reduce the false positive rate in most circumstances. The only exception is in Experiment 8, where both extensions were observed to achieve the same result when m was increased to 640K. This is an extreme case, where m is sufficiently large, coincidental hits will not occur and thus false positives become zero. However, this is also the case where a Bloom filter is degenerated into an ordinary hash table.

Since there is no benefit at all to use Bloom filters as ordinary hash tables, it appears that we can claim that **the refined extension in practice will have less false positives than the intuitive extension in all realistic cases, given the same configuration**. This also implies that with less storage requirement (i.e. smaller m) or less computation (i.e., smaller k) than demanded by the intuitive extension, the refined extension can achieve the same false positive rate.

We also calculated the false positive rate in Experiments 4-8 by dividing the number of distinct elements having an erroneous count with the number of distinct elements inserted into the filter. All the above observations still apply.

Another observation is that for both extensions, when k is fixed, the false positive rate decreases as m grows in proportion to n. This is because there will be less coincidental hits when the size of the filter is increased. However, the false positive rate in both extensions (of a fixed size m) does not necessarily decrease as k increases.

In most of our simulations, the reduction rate in false positives achieved by the refined extension increases as k increases, when the filter size m is fixed; the reduction rate also increases as the filter size increases, when k is fixed. However, both do not hold in general.

The largest reduction rates are observed when the number of insertions for each element is uniformly distributed, and the elements are inserted into the filter in a random order (i.e. in Experiment 4). In the best case, the refined extension has reduced the false positive rate by an order of about 18.

As shown in Experiments 1-5, **the order in which a sequence of elements is inserted can significantly affect the false positive rate in the refined extension, while it has no impact at all for the intuitive extension**, which performs the same in Experiments 1-3 as well as in Experi-

Filter	$k=4$		$k=6$		$k=8$	
size m	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.
80K	5.612E-3	7.312E-4	4.069E-3	6.433E-4	4.213E-3	6.214E-4
160K	5.068E-4	2.295E-4	1.587E-4	1.243E-4	7.710E-5	8.629E-5
320K	3.450E-5	5.692E-5	3.100E-6	1.733E-5	3.000E-7	5.469E-6

(a)

Filter	Reduction Rate		
size m	$k=4$	$k=6$	$k=8$
80K	4.259	5.294	6.046
160K	4.681	5.952	7.375
320K	5.391	8.290	15.000

(b)

Table 4. The false positive rates in $Experiment\,2$ for the intuitive extension is the same as in Experiment 1. (a) shows improved result in the refined extension, and (b) shows reduction rate achieved by the refined extension.

Filter	$k=4$		$k=6$		$k=8$	
size m	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.
80K	1.875E-2	1.392E-3	1.538E-2	1.266E-3	1.707E-2	1.292E-3
160K	1.789E-3	4.265E-4	6.278E-4	2.378E-4	3.482E-4	1.871E-4
320K	1.350E-4	1.160E-4	1.630E-5	4.080E-5	2.700E-6	1.621E-5

(a)

Filter	Reduction Rate		
size m	$k=4$	$k=6$	$k=8$
80K	1.275	1.401	1.493
160K	1.326	1.505	1.633
320K	1.378	1.577	1.667

(b)

Table 5. The false positive rates in $Experiment\,3$ for the intuitive extension is the same as in Experiment 1. (a) shows improved result in the refined extension, and (b) shows reduction rate achieved by the refined extension.

Filter	$k=4$		$k=6$		$k=8$	
size m	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.
80K	2.019E-2	1.734E-3	1.723E-2	1.554E-3	1.964E-2	1.608E-3
160K	1.955E-3	5.295E-4	7.178E-4	2.990E-4	4.059E-4	2.440E-4
320K	1.530E-4	1.426E-4	1.733E-5	4.976E-5	2.662E-6	1.838E-5

(a)

Filter	$k=4$		$k=6$		$k=8$	
size m	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.
80K	5.381E-3	7.781E-4	3.231E-3	5.685E-4	2.929E-3	5.193E-4
160K	4.491E-4	2.140E-4	1.096E-4	9.725E-5	4.738E-5	6.371E-5
320K	3.179E-5	5.712E-5	1.611E-6	1.268E-5	1.501E-7	3.049E-6

(b)

Filter	Reduction Rate		
size m	$k=4$	$k=6$	$k=8$
80K	3.753	5.331	6.706
160K	4.352	6.550	8.567
320K	4.813	10.752	17.733

(c)

Table 6. The false positive rates in $Experiment\,4$: (a) for the intuitive extension; (b) for the refined extension; (c) reduction rate achieved by the refined extension.

Filter	$k=4$		$k=6$		$k=8$	
size m	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.
80K	5.982E-3	8.812E-4	3.952E-3	7.073E-4	3.810E-3	6.532E-4
160K	5.211E-4	2.511E-4	1.446E-4	1.266E-4	6.395E-5	8.661E-5
320K	3.622E-5	6.495E-5	2.853E-6	1.851E-5	2.402E-7	4.379E-6

(a)

Filter	Reduction Rate		
size m	$k=4$	$k=6$	$k=8$
80K	3.375	4.359	5.155
160K	3.751	4.964	6.348
320K	4.224	6.074	11.083

(b)

Table 7. The false positive rates in $Experiment\,5$ for the intuitive extension is the same as in Experiment 4. (a) shows improved result in the refined extension, and (b) shows reduction rate achieved by the refined extension.

ments 4-5. In other words, the insertion order can have a significant impact on the rate of false positive reduction that can be achieved by the refined extension.

The frequency that each element is inserted, i.e. the distribution of c_i, can also have an impact on the rate of false positive reduction. The comparison of reduction rates in Experiment 4 (c_i: uniformly distributed over [0,20]) and Experiment 6 (c_i: Poisson with $\lambda = 10$) shows this. Experiment 8 (c_i: uniformly distributed over [0,40]) vs. Experiment 7 (c_i: Poisson with $\lambda = 20$) is another good illustration.

In all the experiments, we in fact allocate 6 bits to each cell so that we can compare the counter growth in both extensions. The observed counter growth in the refined extension is much slower than in the intuitive extension for each configuration in each experiment[4]. For example, no cells in the refined extension reach the count limit 63 after all the insertions are done, whereas there are many in the intuitive extension. The number of cells with a count larger than 20 in the intuitive extension can be more than 200 times that in the refined one. This implies that false positives in a DCC implementation enhanced by the refined extension can be both

4 Due to the space limit, tables showing the differences are omitted here.

Filter	$k = 4$		$k = 6$		$k = 8$	
size m	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.
80K	2.394E-2	1.643E-3	2.155E-2	1.563E-3	2.546E-2	1.635E-3
160K	2.374E-3	5.269E-4	9.459E-4	3.145E-4	5.655E-4	2.521E-4
320K	1.862E-4	1.452E-4	2.608E-5	5.405E-5	4.178E-6	2.013E-5

(a)

Filter	$k = 4$		$k = 6$		$k = 8$	
size m	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.
80K	1.086E-2	1.036E-3	7.966E-3	8.512E-4	8.282E-3	8.500E-4
160K	9.937E-4	3.134E-4	2.917E-4	1.565E-4	1.536E-4	1.151E-4
320K	7.208E-5	8.158E-5	7.613E-6	2.494E-5	9.843E-7	8.584E-6

(b)

Filter	Reduction Rate		
size m	$k = 4$	$k = 6$	$k = 8$
80K	2.203	2.706	3.075
160K	2.389	3.242	3.681
320K	2.583	3.426	4.245

(c)

Table 8. The false positive rates in $Experiment\,6$: (a) for the intuitive extension; (b) for the refined extension; (c) reduction rate achieved by the refined extension.

Filter	$k = 4$		$k = 6$		$k = 8$	
size m	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.
80K	2.392E-2	1.576E-3	2.155E-2	1.524E-3	2.549E-2	1.590E-3
160K	2.373E-3	5.169E-4	9.452E-4	3.002E-4	5.683E-4	2.418E-4
320K	1.866E-4	1.405E-4	2.555E-5	5.170E-5	4.735E-6	2.216E-5

(a)

Filter	$k = 4$		$k = 6$		$k = 8$	
size m	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.
80K	1.365E-2	1.174E-3	1.038E-2	1.007E-3	1.109E-2	1.045E-3
160K	1.254E-3	3.553E-4	4.058E-4	1.913E-4	2.067E-4	1.361E-4
320K	9.578E-5	9.939E-5	1.007E-5	3.038E-5	1.720E-6	1.285E-5

(b)

Filter	Reduction Rate		
size m	$k = 4$	$k = 6$	$k = 8$
80K	1.752	2.076	2.298
160K	1.892	2.329	2.749
320K	1.949	2.538	2.725

(c)

Table 9. The false positive rates in $Experiment\,7$: (a) for the intuitive extension; (b) for the refined extension; (c) reduction rate achieved by the refined extension.

smaller and less likely to occur at the same time than in a similar system enhanced by the intuitive extension.

It is intriguing that the refined extension has performed so differently in Experiments 1-3: it has the best performance in Experiment 2 but the worst in Experiment 3. Although the results appear to be counter-intuitive, they are in fact reasonable as a careful study reveals in the following.

Filter	$k = 4$		$k = 6$		$k = 8$	
size m	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.
80K	2.205E-2	1.740E-3	1.946E-2	1.621E-3	2.261E-2	1.741E-3
160K	2.155E-3	5.596E-4	8.327E-4	3.309E-4	4.810E-4	2.587E-4
240K	4.903E-4	2.565E-4	1.057E-4	1.217E-4	3.133E-5	6.818E-5
320K	1.704E-4	1.580E-4	2.136E-5	5.376E-5	3.146E-6	2.048E-5
640K	1.104E-5	4.113E-5	5.302E-7	8.760E-6	0.000E+0	0.000E+0

(a)

Filter	$k = 4$		$k = 6$		$k = 8$	
size m	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.
80K	5.866E-3	8.001E-4	3.586E-3	5.756E-4	3.353E-3	5.255E-4
160K	5.153E-4	2.283E-4	1.207E-4	1.050E-4	5.434E-5	6.651E-5
240k	1.094E-4	1.038E-4	1.439E-5	3.496E-5	2.302E-6	1.283E-5
320K	3.975E-5	6.504E-5	3.332E-6	1.639E-5	3.635E-7	4.467E-6
640K	2.585E-6	1.483E-5	6.564E-8	1.305E-6	0.000E+0	0.000E+0

(b)

Filter	Reduction Rate		
size m	$k = 4$	$k = 6$	$k = 8$
80K	3.759	5.425	6.744
160K	4.183	6.898	8.851
240K	4.481	7.342	13.610
320K	4.286	6.409	8.653
640K	4.270	8.077	-

(c)

Table 10. The false positive rates in $Experiment\,8$: (a) for the intuitive extension; (b) for the refined extension; (c) reduction rate achieved by the refined extension.

Experiment 2 is effectively equivalent to $Round_1$ in Experiment 1 – the elements are inserted sequentially, and each inserted once. Some coincidental hits do not cause false positives in $Round_1$ but they do so in subsequent $Round_2, \ldots, Round_{20}$ in Experiment 1. Interesting enough, a few more false positives turn up in $Round_2$, but no more in any other subsequent $Round_3, \ldots, Round_{20}$. This is why the refined extension has performed slightly better in Experiment 2 than in Experiment 1.

We have also examined the growth of false positives in Experiment 3 by dividing the randomized insertion sequence (200,000 insertions) evenly into 20 chunks. False positives were noted once each chunk has been inserted. Since elements in each chunk are inserted into the filter different number of times, new false positives have been observed for each chunk. This echoes an observation discussed earlier: the frequency that each element is inserted matters. All this explains why the refined extension has performed worse in Experiment 3 than in Experiment 1.

Finally, it is worthwhile to note that in some simulations, the standard deviation of the false positive rate is large compared to its mean. Examples include all configurations with

$m = 320K$ in Experiment 1. We have examined all these cases, and found that this is really due to the fact that a majority of the 1,000 round simulations produced no false positives while a minority did. (Histograms showing this fact are omitted here, due to the space limit.) Therefore, such an occurrence of large standard deviations is in fact a feature, not a bug!

6. Conclusion and future work

The main contributions of this paper are as follows. First, we have shown that Bloom filters and their variants can significantly enhance two collaborative spam detection systems. Bloom filters have not hitherto been used for purposes such as we have proposed.

Second, we have identified some new Bloom filter tricks, including 1) a novel notion of "utilisable size" of the Bloom filters, and of "global coincidental hits" and "local coincidental hits" in the filters, and 2) a new Bloom filter variant, which supports counting, heuristics that reduce counting errors by addressing both global and local coincidental hits, and an innovation that reduces its storage cost.

Third, we have performed a simulation study to show that our new variant can effectively reduce counting errors occurred in an intuitive variant of the Bloom filter, unless both are degenerated into an ordinary hash table. This simulation study also has furthered our understanding of these two variants. For example, the frequency with which each element is inserted matters for the rate of error reduction achieved by our new variant. The order in which a sequence of elements is inserted can have a significant impact on the error rates in our variant, but it has no such effect at all in the intuitive variant.

Our ongoing and future work include 1) to estimate, with empirical data, f_{dcc} in a DCC implementation enhanced by our new Bloom filter variant, and 2) to empirically evaluate other performance changes that this variant introduces to the DCC system, e.g. average speed for signature queries. Since such a Bloom filter variant can be applied to applications where it is relevant to support fast membership testing and distributed counting with controllable inaccuracy, we are also interested in identifying its other novel applications in computer security.

Acknowledgement

We thank Isi Mitrani for helping with our probabilistic analysis and simulation design, and thank Ross Anderson, Zoe Andrews, Feng Hao, Brian Randell, Robert Stroud and anonymous reviewers for valuable comments. Will Ng of the Chinese University of Hong Kong pointed our attention to collaborative spam detection schemes.

References

[1] B Bloom. Space/time trade-offs in hash coding with allowable errors. Commun. ACM, 13(7):422-426, 1970.

[2] T Cormen et al. "Introduction to Algorithms (2nd ed.)". The MIT Press, 2002.

[3] Distributed Checksum Clearinghouse, available at `http://www.rhyolite.com/anti-spam/dcc/`.

[4] Li Fan et al, "Summary cache: a scalable wide-area web cache sharing protocol", IEEE/ACM Transactions on Networking, Volume 8, Issue 3, June 2000. pp 281 - 293

[5] Donald E. Knuth. "The Art of Computer Programming", Vol.2, third Edition, Reading, Massachusetts: Addison-Wesley, 1997.

[6] I. Mitrani, Simulation Techniques for Discrete Event Systems, Cambridge University Press, 1982 (Reprinted, 1986).

[7] Vipul Prakash, Razor, available at `http://razor.sourceforge.net/`.

[8] Vipul Prakash, Personal Communication, 13 September 2005.

[9] M. V. Ramakrishna, "Practical performance of Bloom filters and parallel free-text searching", Commun. of ACM, vol. 32, no. 10, pp. 1237 – 1239, Oct. 1989.

[10] K Shanmugasundaram et al. "Payload attribution via hierarchical bloom filters", Proceedings of the 11th ACM conference on Computer and Communications Security (CCS'04), October 2004.

[11] Vernon Schryver, Personal Communication, September 2005

[12] E Spafford, "OPUS: Preventing Weak Password Choices", Computers and Security 11(3), pp. 273-278, 1992

[13] J Yan and P L Cho, "Enhancing Signature-based Collaborative Spam Detection with Bloom Filters", Technical Report CS-TR-973, School of Computing Science, Newcastle University, UK. June 2006.