

Network Forensics on Packet Fingerprints

Chia Yuan Cho, Sin Yeung Lee, Chung Pheng Tan, Yong Tai Tan

DSO National Laboratories, 20 Science Park Drive, Singapore 118230
{cchiayua, lsinyeun, tchungph, tyongtai}@dso.org.sg
<http://www.dso.org.sg>

Abstract. We present an approach to network forensics that makes it feasible to trace the content of all traffic that passed through the network via packet content fingerprints. We develop a new data structure called the “Rolling Bloom Filter” (RBF), which is based on a generalization of the Rabin-Karp string-matching algorithm. This merges the two key advantages of space efficiency and an efficient content matching mechanism. This also achieves analytically predictable False Positive Rates that can be controlled by tuning the RBF parameters. Leveraging upon these insights, we have designed and implemented a practical Network Forensic System that gives the ability to reconstruct the sequence of events for post-incident analysis.

1 Introduction

Network forensics is the “capture, recording, and analysis of network events in order to discover the source of security attacks or other problem incidents” [1]. Where the security of an organization is concerned, the role of network forensics is complementary to intrusion detection – where intrusion detection fails, network forensics is useful for obtaining information about the attack, based on which the source of the attack can be identified and the damage can be contained.

Network Forensic Analysis Tools (NFAT) can capture and store all network traffic and provide the analysis engine to the security analyst [2], [3], [4]. Examples of the features provided include trend analysis, content clustering, traffic playback, detection of traffic patterns and anomalous traffic. While many of these features are useful for network monitoring, in many cases, a form of content-based searching would be far more useful. However, this presents two problems – searching through enormous amounts of raw data is extremely time-consuming and inefficient. On top of that, the enormous amount of data imposes huge storage requirements. For an idea of the magnitude this involves, data corresponding to only 4.6 days for 10% usage of a 100-Mbit/s WAN circuit takes up 1TB [5]. Therefore, it is a widespread practice to write over old data, and the forensic data may be gone precisely when it is needed.

Motivated by the above, we believe that a network forensic system should capture the contents of all packets, and store them in compact representations to efficiently support investigative queries. With storage efficiency, it becomes more feasible to keep data for extended periods of time, thus ensuring that attacks can always be traced even if undetected by Intrusion Detection Systems.

Our contribution in this paper is to present an approach to network forensics which ensures that all packets that passed through the network will always leave behind traceable content fingerprints. We present a new data structure, the “Rolling Bloom Filter” (RBF), which allows string matching to be efficiently performed on content fingerprints by generalizing the Rabin-Karl string matching algorithm. This enables analytically predictable False Positive Rates that can be controlled by tuning the RBF parameters. Leveraging upon these insights, we implement a practical network forensic system that gives the ability to reconstruct the sequence of events for post-incident analysis.

The next section presents the background and some related work. We develop the generalized Rabin-Karp algorithm in Section 3, and show how it can be used for the RBF. In Section 4, we present the performance analysis and optimization of the RBF. The RBF is then used to build a Network Forensic System, the architecture of which is shown in Section 5 and the implementation results are presented in Section 6. We then conclude this paper in Section 8 after a discussion on future work in Section 7.

2 Background

2.1 Bloom Filter Preliminaries

The Bloom Filter is a space-efficient randomized data structure for supporting set membership queries [6]. A Bloom Filter is described as an m bits array, initialized by setting all bits to ‘0’ to denote an empty set. To insert an element, the element is hashed with k independent hash functions, for which each hash output gives an index in the bit array to be set to ‘1’. Thus, a queried element belongs to the set only if all k bits are set to ‘1’.

The Bloom Filter possesses a number of interesting properties: space efficiency, constant $O(k)$ query complexity, zero False Negatives, and controllable False Positive Rates. After n elements are inserted, the False Positive Rate of the Bloom Filter is

$$FP_o = \left(1 - \left(1 - \frac{1}{m} \right)^{kn} \right)^k \approx \left(1 - e^{-\frac{kn}{m}} \right)^k. \quad (1)$$

The False Positive Rate is minimized when $k = \ln 2 \times m/n$, with optimized $FP_o = (1/2)^k = (0.6185)^{m/n}$. There are other standard Bloom Filter operations which can be useful in practical applications – if too few elements are inserted, a Bloom Filter can be halved in size. This is realized by simply taking an OR between the first half of the Bloom Filter and the second half [7]. In addition, if the Bloom Filter density is less than 1/2, further space efficiency is gained through compression [8].

2.2 Applications of Bloom Filters to Network Forensics

To our best of knowledge, the first application of Bloom Filters to network forensics was the Source Path Isolation Engine (SPIE) [9], where it was proposed that each

router stored the digest of all forwarded packets into Bloom Filters. In the event of an attack, each router would be queried to map out the path of the attack, so malicious packets could be traced to their origin. However, packet digests were constructed by hashing the unique fields in the IP packet header together with a predetermined number of bytes from the payload. This means that this scheme supported only the tracing of packets rather than packet contents.

The tracing of packet contents was introduced using Hierarchical Bloom Filters (HBF) in [10]. Content matching was possible because the packet payload was parsed in fixed-size blocks, and the index of each block was appended before insertion into the Bloom Filter. The HBF attempts to reduce False Positives by *aggregating* the results of individual Bloom Filter queries hierarchically. To illustrate this, Fig. 1 shows the insert and query mechanisms for a 3-level HBF with 4-byte unit block size.

Fig. 1 shows that 7 blocks are formed from the 16-byte portion of the content string for the insert operation. What is straightforward is that the same 16-byte substring would match all 7 blocks when used as a query. However, *all* other 16-byte substrings fall into the worst case. For example, consider the substring “BCDE ... NOPQ”. Due to difference in block alignment, only 4 out of 7 blocks can be matched even after trying alignment shifts. This effect gives variable effective False Positive Rates for the HBF, with common worst case performance. Also, storage is inefficient because only about 1/2 the number of inserted blocks are used in the worst case.

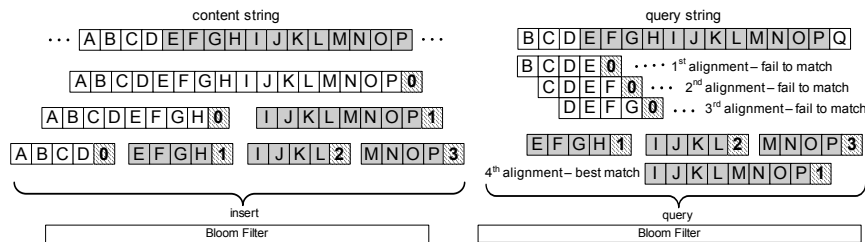


Fig. 1: The HBF with 3 layers of hierarchy, where shaded regions indicate matched portions of the content string for the given query string.

2.3 Rabin-Karp Hash-based String Matching

The Rabin-Karp [11], Boyer-Moore [12], and Knuth-Morris-Pratt [13] string matching algorithms are known to be the most efficient, but we chose the Rabin-Karp because upon generalization, it can be used with Bloom Filters for storage efficiency.

Given a content string of length p and a query string of length q , where $p \geq q$, the Rabin-Karp algorithm slides a window of length q through the content string and compares each hash value against that of the query string. The key to this is the “rolling hash function” [11], where the hash value for each window position can be used to efficiently compute the hash value for (i.e. rolls over to) the next window position. One way to achieve this is via modulo hashing, where a q -byte query string is represented as a number of base b and modulo L , such that b is large and b and L are relatively prime. Since the rolling hash function computes each hash in constant time, the complexity is $O(p)$ and $O(pq)$ for the average and worst case respectively.

3 String Matching on Content Fingerprints

In this section, we develop the Rolling Bloom Filter (RBF), a space-efficient data structure that implicitly performs generalized Rabin-Karp string matching.

3.1 Generalized Rabin-Karp Algorithm

Viewing the ordered series of hash values as fingerprints, our objective is to store the fingerprints of content strings into Bloom Filters so that the efficient Rabin-Karp algorithm can be adapted to use Bloom Filters. This merges the storage space efficiency of Bloom Filters with the string matching efficiency of Rabin-Karp, making it ideal for network forensic applications. More importantly, the Rabin-Karp algorithm inspired us to conceive a technique to aggregate individual Bloom Filter queries *linearly* instead of *hierarchically* (in the HBF). This achieves analytically predictable effective False Positive Rates that can be controlled by tuning the RBF parameters. We show how these advantages can be achieved by first generalizing the Rabin-Karp algorithm.

We generalize the Rabin-Karp algorithm by generalizing: (i) the length of the hash window, w , and (ii) the delta shift for the window at each step, d , where w and d are such that $w \geq d$ and $w \leq q - d + 1$. In the Rabin-Karp algorithm, $w = q$ and $d = 1$ respectively, i.e. the lengths of the query string and hash window are the same, and the window position increments at each step. The generalized algorithm is as follows:

```
1 generalizedRabinKarp(query[0..q-1], content[0..p-1]) {
2   for(s = 0; s < d; s++) //try alignment shifts
3     for(i = 0; i <= p-q; i += d) { //content substrings
4       for(j = s, t = i; j <= q-w; j += d, t += d)
5         if(h(query[j..j+w-1]) != h(content[t..t+w-1]))
6           return NOT_FOUND;
7       if(query[0..q-1] == content[i..i+q-1])
8         return i;
9       else return NOT_FOUND;
10    }
11 }
```

Fig. 2 illustrates the generalized Rabin-Karp algorithm with $d = 2$ and $w = 4$, so that there are also 7 blocks within the 16-byte portion of the content string as in the HBF example. This example reveals the *first key significance* of our generalization. It is clear from Fig. 2 that *any* 16-byte query which is a substring of the content string will *always* match *at least* 6 out of 7 blocks, an advantage over the highly variable number of matched blocks in the HBF. The comparison of False Positive Rates will be presented in sub-Section 4.2

Note also that the complexity of the generalized algorithm can be shown to be $O(p)$ and $O(pq)$ for the average case and worst case respectively, similar to the original algorithm. The key is that the innermost loop makes only $1/(1 - FP_0)$ iterations on average, where FP_0 is the collision probability.

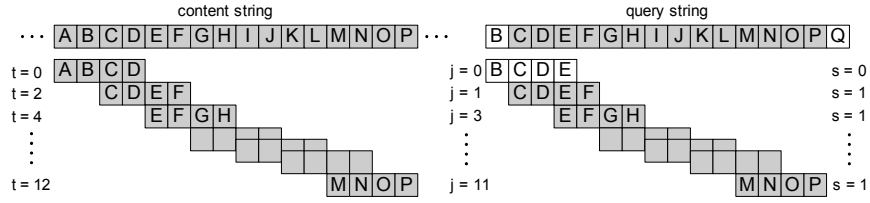


Fig. 2. Illustration of generalized Rabin-Karp with $w = 4$ and $d = 2$. The shaded (unshaded) regions indicate matched (unmatched) regions through hash comparisons

3.2 RBF Primitives

The Rolling Bloom Filter (RBF) is a space-efficient data structure that stores content string fingerprints into Bloom Filters to support approximate string matching. A RBF is described by a set of defined parameters $\{m, n, k, w, d, N\}$ and is represented by a series of N Bloom Filters, where each i th Bloom Filter of $m_i \leq m$ bits stores the i th overlapping fragment of any content string, $\text{content}[di \dots di + w - 1]$ such that $0 \leq i \leq N - 1$. To accommodate varying string lengths, the number of Bloom Filters, N , grows with the maximum length of any inserted content string according to the relationship $N = 1 + \lfloor (p_{\max} - w) / d \rfloor$, where p_{\max} is the maximum length of any inserted content string. The RBF is considered “full” after n content strings are inserted.

Initializing the RBF is the same as initializing a Bloom Filter – each i th Bloom Filter initially takes up $m_i = m$ bits and all bits are initialized to zero. The RBF supports three other primitives – Insert, Query and Optimize.

Insert

To insert a string into the RBF, insert each i th overlapping fragment $\text{content}[di \dots di + w - 1]$ into the i th Bloom Filter, $0 \leq i \leq N - 1$. In reference to generalized Rabin-Karp, this facilitates hash value comparisons in Line 5 of the algorithm.

Query

Generalized Rabin-Karp is the underlying algorithm of the Query primitive, with some modifications. Unlike the generalized algorithm, the RBF does not keep the original content strings for exact string matching. Therefore, the RBF supports only *approximate string matching*. In reference to Fig. 2, we note that a query string can have an unmatched length of at most $(d - 1)$ at each end of the matched substring. Thus, an approximately match is a substring of length l such that $q - 2(d - 1) \leq l \leq q$.

Optimize

This optimizes the RBF size just prior to storage. This primitive is used after n strings are inserted into the RBF, i.e. when the RBF is “full”. Since content strings can be of any length, each i th Bloom Filter in the RBF may contain too few elements n_i , i.e. $n_i \leq n$, and so $m/n_i \geq m/n$. This means the bits per element ratio m/n_i for the i th Bloom Filter may be higher than desirable for storage efficiency. The Optimize primitive optimizes the size of each i th Bloom Filter m_i by iteratively halving its size while doubling its bits per element ratio till $m/2n \leq m_i/n_i \leq 3m/2n$ so that $E[m_i/n_i] = m/n$.

4 Performance Analysis and Optimization

4.1 Performance Metrics

False Positive Rate, FP_{RBF}

In reference to the generalized Rabin-Karp algorithm in Section 3.1, the algorithm makes a string match attempt at each fixed position (s, i) with False Positive Rate $FP_{s,i}$. We approach the problem by first determining $FP_{s,i}$.

Let us denote the outcome events of each individual Bloom Filter query as $\{\text{'X': negative result, 'A': positive result due to actual existence of the string within the hash window, and 'F': positive result due to False Positive}\}$. Thus, each string match attempt produces a series of events which terminates with an 'X' if the string is not matched, or a series of any combination of 'A' and 'F' if the string is successfully matched. For example, if a string requires 6 individual queries, "A, X" is unmatched and any one of the following: "A, A, A, A, A, A", "F, F, F, F, F, F" or "F, A, F, F, A, A" indicates a successful match. Combinations of 'A' and 'F' contribute to $FP_{s,i}$ and are undesirable. The probability $P(F) = FP_o$ can be tuned through the Bloom Filter parameters $\{m, n, k\}$. Further, in uniformly distributed content, $P(A) = 1/2^{8w}$ and random occurrences of 'A' vanish as the hash window length w increases. This makes $P(F) \gg P(A)$, and so $FP_{s,i}$ is dominated by the series of 'F' events.

In reference to the generalized Rabin-Karp algorithm, the minimum number of positive events required for a successful string match is $\lfloor (q-w+1)/d \rfloor$. Thus, if $P(F) \gg P(A)$, the False Positive Rate for the string match attempt at each fixed (s, i) is

$$FP_{s,i} = (FP_o)^{\lfloor \frac{q-w+1}{d} \rfloor}. \quad (1)$$

As (s, i) varies within its entire space of possible values, at most $(p_{max} - q + d)$ string match attempts are made. Thus the probability that the RBF query *does not return a False Positive* is expressed as

$$(1 - FP_{RBF}) = [1 - FP_{s,i}]^{p_{max} - q + d} = \left[1 - (FP_o)^{\lfloor \frac{q-w+1}{d} \rfloor} \right]^{p_{max} - q + d}. \quad (2)$$

Finally, by expanding the right hand side via Binomial Theorem, the effective False Positive Rate of each RBF query is approximately

$$FP_{RBF} \approx (p_{max} - q + d)(FP_o)^{\lfloor \frac{q-w+1}{d} \rfloor}. \quad (3)$$

Note that any aggregation of Bloom Filter queries may still produce high False Positive Rates if occurrences of event 'A' are not controlled – otherwise, all permutations of 'X' and 'A' events add to False Positives. Indeed, this explains the abnormally high False Positive Rates (beyond worst case) experienced by the HBF on "structured traffic" as reported in the extended version of [10]. Herein lies the *second key significance* of our algorithm. Without requiring extra resources, the hash window length w can be increased (within its constraints) to vanish occurrences of event 'A', so that the aggregation of Bloom Filter queries *must* exponentially reduce False Positive

Rates. This produces analytically predictable False Positive Rates that can be controlled by tuning the RBF parameters.

Data Reduction Factor, D

The Data Reduction Factor quantifies the factor of storage space reduction by the RBF. On average, each content string is stored into $1 + \lfloor (\hat{p} - w) / d \rfloor$ Bloom Filters, where \hat{p} is the mean length of the n content strings inserted into the RBF. Since the mean bits per element ratio is m/n after using the Optimize primitive, we have

$$D = \frac{8\hat{p}}{\left(1 + \left\lfloor \frac{\hat{p} - w}{d} \right\rfloor\right) \left(\frac{m}{n}\right)} \geq \frac{8d}{m/n} \text{ since } w \geq d. \quad (4)$$

As a result of the lower bound, we can set $D_{des} = 8d/(m/n)$ as the *desired minimum Data Reduction Factor* to be achieved by the RBF. Note that our Rabin-Karp generalization allows d to be tuned to achieve the desired Data Reduction Factor.

Minimum Query Length, q_{min}

Since the RBF does not rely on the original content strings to resolve hash collisions, the query length will have to be bounded by q_{min} if we wish to achieve a desired False Positive Rate. This will be elaborated upon in the following section.

4.2 Performance Analysis

Given the desired minimum Data Reduction Factor D_{des} , our objective is to study the RBF performance. We first note that the set of RBF parameters $\{m, n, k, w, d, N\}$ can be reduced since it contains many non-independent variables. From Section 2.1, the False Positive Rate for each Bloom Filter, FP_o , is optimized at $k = \ln 2 \times m/n$. Given $D_{des} = 8d/(m/n)$, we can also determine d from m/n . Finally, N need not be predetermined since it grows with the maximum content string length p_{max} . The reduced set of RBF parameters that change independently is thus simply $\{k, w\}$.

We now analyze the performance of the RBF using some numerical examples. Let us first assume that w is sufficiently large and the claim $P(F) \gg P(A)$ in Section 4.1 is true so that the RBF False Positive Rate is given by Eq. (3). Note that otherwise, the False Positive Rate is higher than given in Eq. (3), which then constitutes the lower bound. The relationship between FP_{RBF} and the query length q is shown in Fig. 3 as k and D_{des} vary at $w = 2d$. By drawing a horizontal line at desired $FP_{RBF} = 10^{-5}$ we obtain the minimum query length $q_{min} = 167$ required to achieve $FP_{RBF} = 10^{-5}$. It is also apparent that the lowest FP_{RBF} at any q and D_{des} is at $k = 1$, where the individual Bloom Filter False Positive Rate is $P(F) = FP_o = 1/2$.

In practice, $P(A)$ depends on the nature of the content. It is likely that $P(A) = 1/2^{8w}$ for uniformly distributed content can never be found in natural text. However, even for a 26 – alphabet character set, $P(A) = 1/26^w$ if all characters are independent or $P(A) = 1/26$ if all characters within the window are completely dependent. Since FP_{RBF} is optimized at $k = 1$ giving $P(F) = 1/2$, then $P(F) \gg P(A)$ is satisfied for all

cases by choosing $k = 1$. Finally, choosing $w = 2d$ (at least) is required to vanish random occurrences of event ‘A’ and for Eq. (3) to be valid.

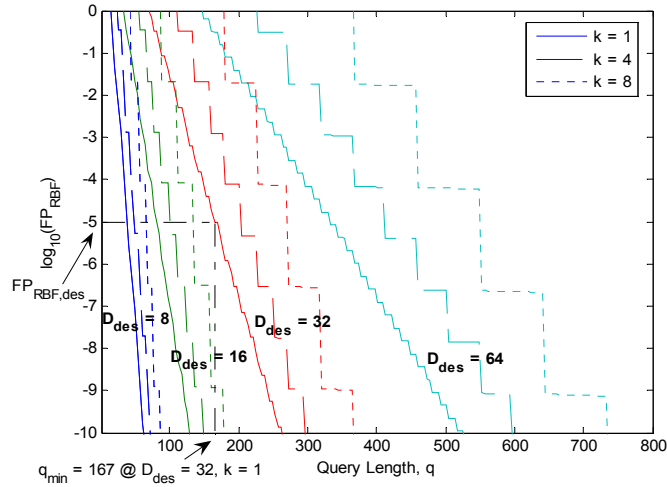


Fig. 3. Log Scale variation of the FP_{RBF} with q , at different D_{des} and m/n .

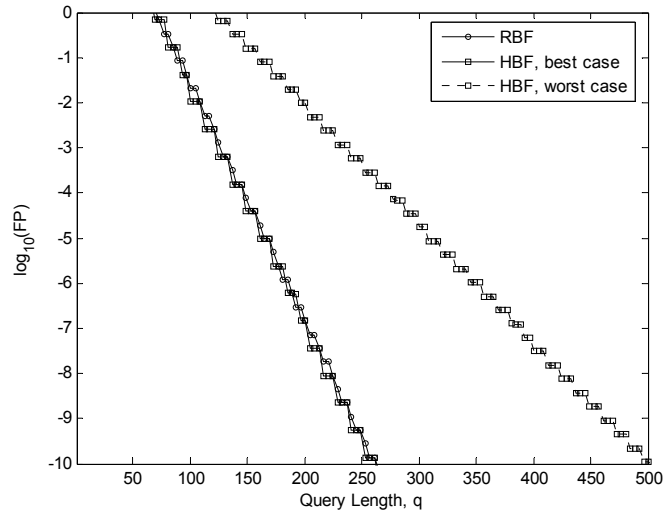


Fig. 4. Comparison of False Positive Rates of the RBF and the HBF.

We now compare the performance of the RBF with the Hierarchical Bloom Filter (HBF). The False Positive Rate of the HBF can be evaluated following the principles presented in this paper. Fig. 4 compares the effective False Positive Rates of the RBF in comparison with the HBF, both using the same parameters to give the same desired Data Reduction Rate $D_{des} = 32$. It is evident that the RBF consistently achieves False Positive Rates same as or near to the *best case* for the HBF.

5 The Network Forensic System

The Network Forensic System supports content-based queries on all traffic that has passed through the network within the monitoring period. Given a file binary or fragment as the query input, the Network Forensic System returns the “information of interest” on all occurrences of that input, or a null reply if the input has not been encountered. This includes the time of occurrence, source and destination IP addresses, MAC addresses, source and destination ports, traffic statistics and of course, if possible, references to more information about the content.

Depending on the desired level of security, the Network Forensic System may be deployed at all hosts throughout the network, or only at the network perimeter, or a hybrid between the two, e.g. on shared segments or subnet gateways. At each monitoring point, data is stored in archive units, each of which consists of three core components – a Meta-Fingerprint Filter, a Micro-Fingerprint Filter database, and a Connection Endpoint Table. These are illustrated in Fig. 5.

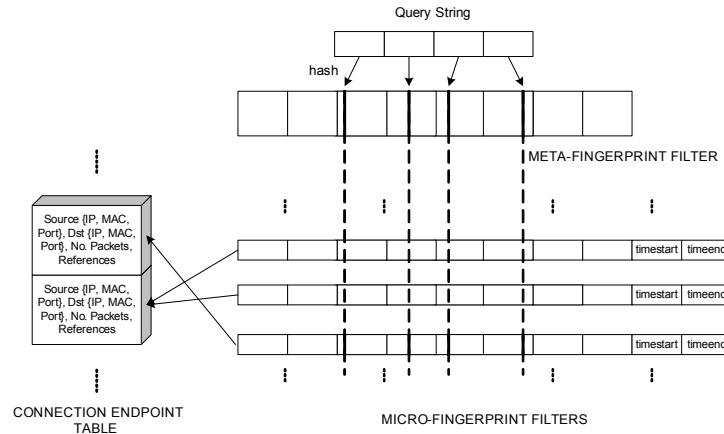


Fig. 5. Core components of an archive unit in the Network Forensics System

The Meta-Fingerprint Filter is a RBF with predefined parameters $\{m, n, k, w, d\}$. The RBF parameters are chosen such that the performance requirements are satisfied as outlined in Section 4.2. The Meta-Fingerprint Filter is the front-end filter for input queries – it allows null replies to be quickly returned if a string is not found, the most frequent case in practice. A Micro-Fingerprint Filter is a small RBF that stores more precisely the fingerprints of a small group of n_μ packets referencing the same entry in the Connection Endpoint Table, where $1 < n_\mu < 10$.

When a match occurs in the Meta-Fingerprint Filter, a precise match is attempted on the database of Micro-Fingerprint Filters, where a match indexes the information entry on the Connection Endpoint Table. For space efficiency, the Connection Endpoint Table stores aggregated information such as the source and destination IP addresses, MAC addresses, source and destination ports, and traffic statistics; timestamps are appended to the Meta-Fingerprint Filter instead. By doing so, the same

entry in the Connection Endpoint Table can be referenced by a large number of Meta-Fingerprint Filters, thereby achieving space efficiency.

If each Micro-Fingerprint Filter uses the same RBF parameters $\{m, n, k, w, d\}$, the mean number of False Positive entries returned by the Micro-Fingerprint database given a matched Meta-Fingerprint can be evaluated as

$$\mu = \frac{n}{n_{\mu}} (FP_o)^{\lfloor \frac{q-w+1}{d} \rfloor}. \quad (5)$$

6 Implementation Results

We implemented a Network Forensic System with parameters as indicated in Table 1. Each archive unit took up about 70MB, compared to near to 1GB from raw packet traces. Since redundancy exists even in the RBF, each archive unit could be further compressed using Lempel-Ziv Markov Chain (LZMC) compression to around 55MB.

Table 1. Performance, RBF parameters and capacity of Network Forensic System

	Macro-Fingerprint Filter	Micro-Fingerprint Filters	Archive Unit
Performance	$q_{\min} = 167$		
	$D_{des} = 32, FP_{RBF} = 10^{-5}$		$D \approx 16$
	$\mu = 1.3 \times 10^{-3}$		
RBF Parameters	$m/n = 1/\ln 2, k = 1, w = 12, d = 6$		
Capacity (No. packets)	$n = 2^{20}$	$n = 2^{20}$ (all Filters)	$n = 2^{20}$
	-	$n_{\mu} = 8$ (each Filter)	

Table 2. Summary of results returned from queries to the Network Forensic System

	Approx. Time	Source		Destination		Sig-nature	Monitor Point	Pkts
		IP	Port	IP	Port			
1	17:08:44 ~ 17:08:44	x.x.x.133	1083	x.x.x.241	445	Shell-code	Machine 1 & 2	11
2	13:39:03 ~ 14:10:08	x.x.x.133	54321	x.x.x.131	1175	Cmd-banner	Machine 1	72
3	13:38:49 ~ 13:50:49	x.x.x.131	1174	x.x.x.133	445	Shell-code		16
4	12:43:20 ~ 12:43:20	x.x.x.131	1118	x.x.x.193	445		Server	1814
5	12:23:56 ~ 12:23:57	x.x.x.194	80	x.x.x.131	1097	Source code	Perimeter	12
6	12:23:41 ~ 12:23:57	x.x.x.194	80	x.x.x.131	1094			13

Machine1: x.x.x.133 Machine2: x.x.x.241 Internal Server: x.x.x.193
External Web Server: x.x.x.194 Attacker: x.x.x.131

We deployed the Network Forensic System in an experimental testbed with monitoring points at the network perimeter, servers, and host machines and simulated an insider attack using a recent publicly available exploit based on MS05-039 [14]. We

simulated the scenario where the attack was not detected by IDSs, but anomalies were discovered later at the victim machine. We suppose an exploit binary was recovered at the victim machine, from which its shellcode is used as the signature query string. Starting from this single host and attack signature, all hosts involved in the attack can be identified and more signatures can in turn be extracted from their recovered files, command banners and protocol banners to paint a complete picture of the attack.

The sequence of events is shown in reverse chronological order in Table 2. The attacker browsed the web, downloaded the exploit source code (rows 5 and 6), and launched the exploit on an internal server as first attempt (row 4). The second attempt was launched on the discovered victim machine, Machine1 (row 3) and was successful (row 2). This time the attacker gained shell, and hours later (row 1), Machine1 was used to attack Machine2 in proxy.

We verified the False Positive Rate of the Micro-Fingerprint Filter database by using real samples of varying lengths to query the Network Forensic System. The variation of μ , the mean number of false positive entries returned, with query length q is shown in Fig. 6. The discontinuities in the measured results with increased query length were due to the lack of False Positive matches at the Meta-Fingerprint Filter level. Actual measurements show that the Network Forensic System perform better than the analytical bound, and this is likely to be due to the fact that not all RBFs were filled up, giving variations in the actual density of RBFs within the database.

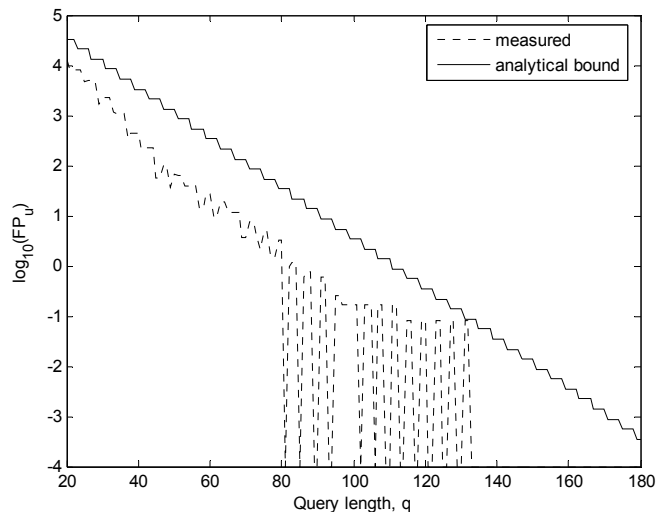


Fig. 6. Log scale variation of μ , the mean number of False Positive entries returned by the Micro-Fingerprint Filter database given a Meta-Fingerprint Filter match.

7 Future Work

At present, False Negatives are possible because a file can be split over multiple packets. This is minimized currently by using multiple different parts of a file as query signature strings. For future work, this can be totally eliminated by extending

the RBF Query primitive to take this into account. Also, since network content is sometimes encoded (e.g. MIME for email), a useful improvement would be to iterate queries over multiple encoders in the Network Forensic System.

8 Conclusion

We have presented an approach to network forensics that makes it feasible to trace the content of all traffic that passed through the network via packet content fingerprints. To achieve this, we have developed the Rolling Bloom Filter, which is based on a generalization of the Rabin-Karp string-matching algorithm. This successfully merged the two advantages of space efficiency and an efficient content matching mechanism. Our technique also achieved analytically predictable False Positive Rates that can be controlled by tuning the RBF parameters. Using these results, we have designed and implemented a practical Network Forensic System that gives the ability to reconstruct the sequence of events for post-incident analysis.

9 References

1. "Network Forensics", searchSecurity.com Definitions, <http://searchsecurity.techtarget.com>.
2. NIKSUN, NetDetector, <http://www.niksun.com>.
3. Computer Associates, eTrust Network Forensics, <http://www3.ca.com>.
4. Sandstorm Enterprises, NetIntercept, <http://www.sandstorm.com>.
5. T. Nisase, M. ItohNetwork, "Forensic Technologies Utilizing Communication Information", NTT Technical Review, Vol. 2, No. 8, Aug 2004
6. B. Bloom, "Space/time tradeoffs in hash coding with allowable errors", Communications of the ACM, 13(7):422--426, 1970
7. A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey", Internet Mathematics, 1(4):485-509, 2004.
8. M. Mitzenmacher, "Compressed bloom filters", Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing, pages 144--150, 2001.
9. A. C. Snoeren, et. al., "Hash-based IP traceback", ACM SIGCOMM, San Diego, California, USA, August 2001, pp. 3--14, 2001.
10. K. Shanmugasundaram, H. Brönnimann, N.D. Memon, "Payload attribution via hierarchical bloom filters", ACM CCS 2004, pp 31-41.
11. R. M. Karp, M. O. Rabin, "Efficient randomized pattern-matching algorithms", IBM Journal of Research and Development 31 (2), 249-260, March 1987.
12. R. S. Boyer and J. S. Moore, "A fast string searching algorithm", Communications of the ACM, 20:762-772, 1977.
13. Knuth D.E., Morris (Jr) J.H., Pratt V.R., "Fast pattern matching in strings", SIAM Journal on Computing 6(1):323-350, 1977.
14. House of Dabus, "Microsoft Windows Plug-and-Play remote overflow universal exploit that is related to MS05-039", <http://www.packetstormsecurity.org/>