Efficient signature based malware detection on mobile devices[1]

Deepak Venugopal[a,2,*] and Guoning Hu[b,2]
[a]*Nokia Inc, 6000 Connection Dr, Irving, TX 75039, USA*
E-mail: deepak.venugopal@nokia.com
[b]*Truveo Inc, An AOL Company, 333 Bush Street, San Francisco, CA 94104, USA*
E-mail: guoninghu@aol.com

Abstract. The threat of malware on mobile devices is gaining attention recently. It is important to provide security solutions to these devices before these threats cause widespread damage. However, mobile devices have severe resource constraints in terms of memory and power. Hence, even though there are well developed techniques for malware detection on the PC domain, it requires considerable effort to adapt these techniques for mobile devices. In this paper, we outline the considerations for malware detection on mobile devices and propose a signature based malware detection method. Specifically, we detail a signature matching algorithm that is well suited for use in mobile device scanning due to its low memory requirements. Additionally, the matching algorithm is shown to have high scanning speed which makes it unobtrusive to users. Our evaluation and comparison study with the well known Clam-AV scanner shows that our solution consumes less than 50% of the memory used by Clam-AV while maintaining a fast scanning rate.

Keywords: Malware, signature detection, mobile and security

1. Introduction

There has been a considerable increase in the use of mobile devices for data services in addition to voice services. New network infrastructures are geared towards enhancing data services to these devices [18,22]. Streaming media on mobile devices and mobile credit card payment are examples of such services [2]. As the mobile network infrastructure continues to grow, the range of functionalities on a mobile handset is increasing as well. Mobile platforms like Symbian Operating Systems, Windows Mobile etc. allow execution of complex applications which were not seen previously on such devices. However, all these developments have made mobile devices an inviting target for hackers and virus writers. Access/destruction of private information stored on mobile devices is a serious concern for users today. Apart from this, spreading worms can have a serious impact on performance of the mobile network. This is due to illegal consumption of bandwidth which can be very damaging since the available bandwidth is still somewhat limited on existing mobile networks. In short, malware in the mobile domain can potentially cause financial losses to both the user and network operator.

[1]A preliminary version of this work was presented at Wicon '06 [33].
[2]This work was done when the author was a part of SMobile Systems Inc.
[*]Corresponding author.

Mobile malware has been steadily increasing over the last two years [9,15]. Even though existing malware are relatively simple compared to those on the PC domain, it is expected that future attacks would inflict heavier damage. Hence, it is important to tackle the problem of mobile malware before it reaches alarming proportions. Since the anti-malware industry for PC is fairly mature, there are a number of general techniques that have been developed to counter malware. But it is important to note the basic differences between a PC and mobile device, especially the constraints on computation power, memory and limited battery resources. The targeted exploits of mobile malware are also significantly different from those on PC due to the differences in operating systems and hardware. For e.g. Majority of mobile devices are based on the ARM architecture [8]. Hence, we need to provide due consideration when using the PC based methods for mobile devices. Some criteria for malware detection on mobile devices include:

- The detection method must use memory and computational resources efficiently and not drain the device battery.
- The detection method must have low false alarm rate i.e. considering a non malware file as malware.
- The detection method must be easy/cost-efficient to update over the wireless network.

There are two general ways of protecting the mobile device. One is to offer protection at the device level and the other is to offer protection at the network level by inspecting packets destined for the device. Device based protection detects and cleans malware including viruses, Trojans and spyware that are installed on the device whereas network based protection looks to detect and prevent intrusions to the network. For the purpose of this study, we look at device based protection that scans files resident on the device for malware.

Since malware detection at the PC level is a well studied problem, some well developed techniques [23] have been developed for PC based anti-malware including:

- Signature based detection
- Heuristic Scanning
- Emulation

Signature detection is based on searching for previously defined virus signatures in input files. A virus signature is a sequence of bytes unique to the virus and not occurring in normal files. Such a signature is usually extracted by a virus expert after its analysis. Manual signature extraction can sometimes be a time consuming process. However, some automatic signature extraction systems [16] have been developed successfully in order to speed up the process of signature extraction. Signature detection is a widely used method on the PC domain due to its simplicity and its ability to yield low false alarms i.e. considering normal files as malware. However, signature detection does have the disadvantage that it cannot detect new viruses for which signatures have not been extracted. Generic signatures can, to a certain extent detect variants of existing viruses. For this, the signatures need to be extracted from the content common to all the variants.

In contrast to signature based detection, heuristic scanning is based on identifying common functionalities/features of viruses. The idea behind heuristic scanning is to detect new viruses based on its common features with existing ones [14,19]. However, it has the disadvantage that it is sometimes prone to false alarms. Since mobile viruses have been surfacing only recently, there are very few mobile virus samples compared to the number of samples available on the PC domain. The lack of sufficient number of virus samples in the mobile domain makes it difficult to extract robust common features and develop heuristic detection techniques based on the available data. If heuristic scanning techniques are developed using limited data, it may be more prone to false alarms or be tuned to the training data i.e. it can only detect viruses of the training set and not generalize very well. This may also require large changes or updates to the feature set very frequently since the feature set is not very stable. Hence, we do not consider the usage of heuristic scanning for mobile devices at this stage.

Emulation is a technique where we allow the virus to run in a virtual environment in order to safely establish the execution pattern of the virus [17]. The major problem with adapting this technique for mobile devices is that it requires a considerable amount of computational power and memory. This method was originally developed for PC based viruses where the constraints on memory and computational resources are not as high as it is on mobile devices. Hence, on constrained devices like mobile phones with limited system resources and running on limited battery power, we don't consider this approach. This method can still be used effectively for mobile malware analysis in remote labs to establish whether a new sample is a virus or not. Other methods can then be used to detect and clean this virus on the actual device.

Keeping in mind the considerations for malware detection on a mobile device, we develop a signature based detection method. It should also be noted that signature detection will remain a part of mobile anti virus systems even if other techniques like heuristic scanning are developed since it facilitates quick detection of known virus types. Here, our aim is to develop a signature matching method such that it has high scanning speed and low memory usage which makes it suitable for mobile devices. Specifically, we use a hash table to store the hash values of signatures to increase scanning speed. Additionally, we aim to eliminate mismatches faster when matching non malicious files by utilizing the probability of occurrence of signature bytes in non malicious content. For the purpose of evaluation, we compare the performance of our scanner against the well-known Clam-AV scanner [31] which utilizes the Aho-Corasick multi pattern matching algorithm [1] for signature based detection. The results on a mobile handset show that our scanner uses up to 50% less memory with a scanning speed slightly better than that of the Clam-AV scanner. The next three sections describe the signature matching method in detail and Section 5 describes the performance of our method on Symbian platform along with the comparison results.

2. Multi pattern matching

As mentioned before, signature based scanning involves matching a set of signatures against an input file. In general, it is a Multi pattern matching problem. Given a set of N signatures, $S = \{S_i\}$ of lengths $L = \{L_i\}$, $1 \leqslant i \leqslant N$, an input sequence of bytes $F = (f_1 f_2 \ldots f_m)$, $0 \leqslant fi \leqslant 255$, $1 \leqslant i \leqslant m$, we need to check if position x exists such that: $1 \leqslant x \leqslant (m - Lq + 1)$, where m is the size of the input file and L_q is the length of the smallest signature in S and x is the start position of a subsequence in F that exactly matches with at least one signature in S.

A naive solution to the multi pattern matching problem involves matching every pattern sequentially against the input. For a file of size m to be matched against Nsignatures each of length n bytes, the worst case running time is $O(N * m * n)$ which is very inefficient. The KMP (Knuth-Morris-Pratt) algorithm [10,30] based on dynamic programming offers significantly better results for single pattern matching. The Rabin-Karp algorithm uses a hashing approach to single pattern matching [26] and can be extended to multi pattern matching with some effort. Its worst case complexity is that of a naive pattern matching method even though practically it has proven to give much better results. The Boyer-Moore algorithm is the most well known single pattern matching solution developed thus far [27]. Several solutions have been proposed to extend the Boyer-Moore algorithm for multi pattern matching [6,35]. One of the drawbacks of the Boyer-Moore based methods is that all of them have a pre-processing phase on the pattern set and store additional tables thereby consuming more memory apart from storage of patterns themselves. The Aho-Corasick method based on finite automata also requires the storage of complete signatures in memory in the form of a tree structure during matching [1]. Hence, if there are a number of long signatures that need to be matched, it may use up more memory resources on a mobile device. All the above algorithms can be applied to any general string matching problem. However, since our solution is specific to signature based detection for mobile, we use the observation that some bytes are less likely to occur in non malicious mobile applications than others (details in Section 5.3). In doing so, we can reduce the number of comparisons during signature matching.

2.1. Hash table

The simplest way to match an input file against N signatures is to match every subsequence in the input file against every signature one by one. This method is computationally very inefficient. One of the well known methods to speed up pattern matching is by building a hash table [30,34]. In this case, each signature is stored at a separate index in the hash table. This index value is computed from the signature itself. For any input subsequence, its index is computed, which points to an entry in the hash table. If there is exactly one signature at each index of the hash table, we ensure that we need to match every input subsequence against one signature instead of all N signatures. One of the key parts to building the hash table is to have an efficient/simple function that, given a sequence computes the index in the hash table for that sequence. For the signatures, the index needs to be computed just once i.e. while building the hash table. However for the input file, the index needs to be computed for every subsequence in the file. Hence, it is essential that the computation of the index be done efficiently. Here, we use a simple method to compute a hash value for a sequence of bytes which we refer to as a filter-hash and use this to obtain the index in the hash table.

For any sequence of bytes $F = (f_1 f_2 \dots f_m)$, $0 \leqslant fi \leqslant 255$, $1 \leqslant i \leqslant m$, we compute a filter-hash value using a continuous subsequence of L' bytes starting from position j as:

$$H(F, j, L') = \sum_{i=1}^{L'} i * fi + j - 1 \tag{1}$$

The index for sequence F is obtained from the filter-hash value computed using Eq. (1) by:

$$h(H(F, j, L')) = \operatorname{mod}(H(F, j, L'), Q), \tag{2}$$

where Q is the size of the hash table.

An important property of the filter-hash value is that for an input file, the filter-hash value can be computed recursively for successive sub-sequences of a common length. The proof for this is shown as:

Initially Let,

$$H(F, 1, L') = \sum_{i=1}^{L'} i * fi, \tag{3}$$

and the sum of L' consecutive bytes initially be,

$$Hs(F, 1, L') = \sum_{i=1}^{L'} fi \tag{4}$$

It can be seen that:

$$H(F, j+1, L') - H(F, j, L') = \sum_{i=1}^{L'} i * fi + j - \sum_{i=1}^{L'} i * fi + j - 1 \tag{5}$$

Simplifying Eq. (5) we have,

$$H(F, j+1, L') = H(F, j, L') + L' * fL' + j - Hs(F, j, L') \tag{6}$$

$$Hs(F, j+1, L') = Hs(F, j, L') - fj + fj + L' \tag{7}$$

Hence, using Eqs (6) and (7) we can obtain the filter-hash values and subsequently the hash table index of consecutive subsequences of length L' in the input file.

One of the essential parameters in a hash table is the size of the hash table, Q. The size of the hash table must be big enough such that each signature can occupy one unique slot in the table. This size is chosen to be greater than the total number of signatures, N. Hence, there are Q slots in the hash table out of which N is occupied by signatures and $Q - N$ is empty. Having a larger number of empty slots influences computational time since this would mean that no signature matching is required when a subsequence hash indexes an empty slot. On the other hand a very large hash table can also increase computational time since we would require more time for memory access in the table. Hence we run experiments with different hash table sizes in order to find the size which minimizes computational time (details in Section 5.3).

2.2. *Signature hash*

Mobile device generally have lower memory resources than PC's with typical system RAM between 8–16 MB [25]. For instance, the Nokia 9210 has 8 MB RAM, the Sony Ericsson P800 has 16 MB RAM. This implies that applications running on mobile devices must be designed to utilize memory efficiently. In order to perform signature matching, we need to store the bytes corresponding to all the signatures in memory. However, as the number and length of the signatures grow, the memory consumption may be very high. In our method, we store a hash value of the signature instead of the signature itself in order to reduce memory consumption. For this, we use a hash value which represents the entire signature. D.J. Bernstein's (DJB) 'times 33' hash [24] posted originally on the Usenet group comp.lang.c is one of the best known hashes for strings and generates a very strong hash value i.e. the probability that different subsequences have the same hash value is extremely low and we consider it negligible for practical cases [3]. Hence, we use the DJB algorithm to hash the entire signature and store it in the hash table. The other advantage of using a DJB hash value is that it generates a hash value of constant size in terms of memory irrespective of the length of the signature.

For any sequence of bytes $F = (f_1 f_2 \ldots f_m)$, $0 \leqslant fi \leqslant 255$, $1 \leqslant i \leqslant m$, we compute a DJB hash value for the continuous sequence of bytes starting from position j of length L' using:

$$D(F, j, L') = (\ldots(((5381 * 33 + fj) * 33 + fj + 1) * 33 + fj + 2) * \ldots) * 33 + fj + L' \tag{8}$$

Unlike the filter-hash value described in the previous section, the DJB hash cannot be computed recursively for successive subsequences i.e. $D(F, j+1, L')$ cannot be computed from $D(F, j, L')$. This means that for every sequence of size L', it takes $O(L')$ time to compute a DJB hash value which is more than the constant time taken for the filter-hash computation. Hence, it is desired that we minimize the number of DJB hash computations during matching.

3. Signature matching

When any subsequence in the input file hashes into a slot (index) in the hash table occupied by a signature, we need to compare the DJB hash value of the signature (stored in the slot) with the DJB hash value of the subsequence (that needs to be computed). In order to make the signature matching efficient, it is necessary to eliminate mismatches before the DJB hash value is computed for the subsequence. For this, we compare the bytes in a small portion of the signature with those in the subsequence. This small portion or set of contiguous bytes in the signature is called the signature cut. For the signature cut to be effective in eliminating mismatches, it must be a sequence of bytes which are less likely to occur in non malicious files. This would ensure that subsequences from non malicious files are likely to be eliminated from further processing (computation of DJB hash) during scanning.

Since the filter-hash value as described in Section 2.1, is also computed for a subsequence of contiguous bytes in the signature, we use the signature cut to compute the filter-hash value. This reduces the amount of information that needs to be stored for each signature. That is, we need to only store the position of the signature cut whereas if the signature cut was different from the filter-hash byte sequence, we would additionally need to store the position of the byte subsequence from which the filter-hash was computed.

3.1. *False alarm rate*

The false alarm rate of a particular sequence of bytes is the probability of occurrence of the sequence in a non malicious file. Since it is difficult to obtain this value analytically, we use the frequency of occurrence in a large number of non virus files to obtain the false alarm rate. For any sequence of bytes $F = (f_1 f_2 \ldots f_m)$, $0 \leqslant fi \leqslant 255$, $1 \leqslant i \leqslant m$, the simplest method to obtain the false alarm rate is to obtain the frequency of occurrence of the entire sequence F in the non virus set. However, the major problem with this method is that it is difficult to obtain accurate probabilities for the entire sequence since it is likely that the entire sequence may not occur at all in the non virus set which is used. However, it is possible to reasonably estimate the probability of the entire sequence. For this, we compute the false alarm rate by utilizing probabilities of smaller subsequences of F. Specifically we use subsequences of size two to estimate the probability. In other words, we utilize the dependency of a byte on one of its prior byte while computing the probability. This can be represented as:

$$P(fm|fm-1\ldots f1) = P(fm|fm-l), \tag{9}$$

where l is a positive integer, which specifies the prior byte that is most relevant to the current byte.

Based on information theory [32], the relevance between two bytes is measured by their mutual information. Therefore, l is the integer such that the mutual information between two bytes separated by $l-1$ bytes is maximized, that is

$$l = \arg\max_{l'} I(l') \tag{10}$$

where

$$I(l) = \sum_a \sum_b P(f_n = a, f_{n-l} = b) \log_2 \frac{P(f_n = a, f_{n-l} = b)}{P(f_n = a)P(f_{n-l} = b)} \tag{11}$$

$I(l)$ is the mutual information between two bytes separated by $l-1$ bytes, which is only dependent on l.

Using Eq. (9),

$$P(f1f2\ldots fm) = \prod_{i=l+1}^{m} P(fi|fi-l) \tag{12}$$

Using Bayes' theorem Eq. (12) can be written as,

$$P(f1f2\ldots fm) = \prod_{i=l+1}^{m} \frac{P(fi, fi-l)}{P(fi-l)}, \tag{13}$$

where the probabilities $P(f_i, f_{i-l}), P(f_{i-l})$ are obtained by counting the number of occurrences of sequence $f_{i-1}f_l$ and f_{i-l} respectively, in a large non malicious data set.

3.2. *Choosing a cut set*

A signature cut is used to compute the index for a signature in the hash table and also to eliminate mismatches. Hence, we have the following criteria while choosing a cut set from the signatures:

1. The number of collisions in the hash table must be minimized
2. The bytes in the cut must occur rarely in non malicious files

With the above considerations, the problem of finding the best signature cut set can be represented as:

Given the signature set $S = \{S_i\}$ $1 \leqslant i \leqslant N$, let $S_{ij}^{L'}$ represent a cut of S_i of length L' starting at position $j, 1 \leqslant j \leqslant Li - L' + 1$, where L_i is length of signature S_i. Let the false alarm rate (computed as described in Section 3.1) of the cut $S_{ij}^{L'}$ be represented by $P(S_{ij}^{L'})$.

We need to obtain a cut set $C = \{S_{ij}^{L'}\}$, such that the following two terms are minimized:

- $\sum\limits_{i=1}^{N} P(S_{ij}^{L'})$ (14)
- $h'(C)$ (15)

where $h'(C)$ represent the number of collisions suffered by members of the cut set i.e. the number of times $h(H(S_i, j, L')) = h(H(S_k, p, L'))$ for $i \neq k$ and $S_{ij}^{L'}, S_{kp}^{L'} \in C$.

The problem of finding an optimal cut set is NP-hard and is reducible to a knapsack problem [21,29]. However, finding an optimal cut set is critical in order to achieve gains in terms of scanning speed. Also, since this process is done offline, speed or efficiency is not criterion that we consider while choosing the cut set. Hence we use an exhaustive search to give us the most optimal cut set.

In order to find such a cut set, we iteratively choose a combination of cuts such that one cut is chosen from every signature and discard this cut if the average value of Eqs (14) and (15) is higher than a previously chosen combination. In this way, when the algorithm terminates we have a cut set with the lowest average value of collisions and false alarm rate. If the value of $h'(C)$ in the resulting cut set is greater than 0, it means a collision exists among signature cuts. In this case, we chain the corresponding signatures in the hash table using a linked list. For any input subsequence hashing into this slot, we traverse through the list matching the subsequence with all the signatures that have hashed into the common slot.

3.3. *Length of the signature cut*

Since all the bytes of a signature cut are stored in memory during matching, the length of the signature cut must be much smaller compared to the length of the signature to achieve gains in terms of low memory

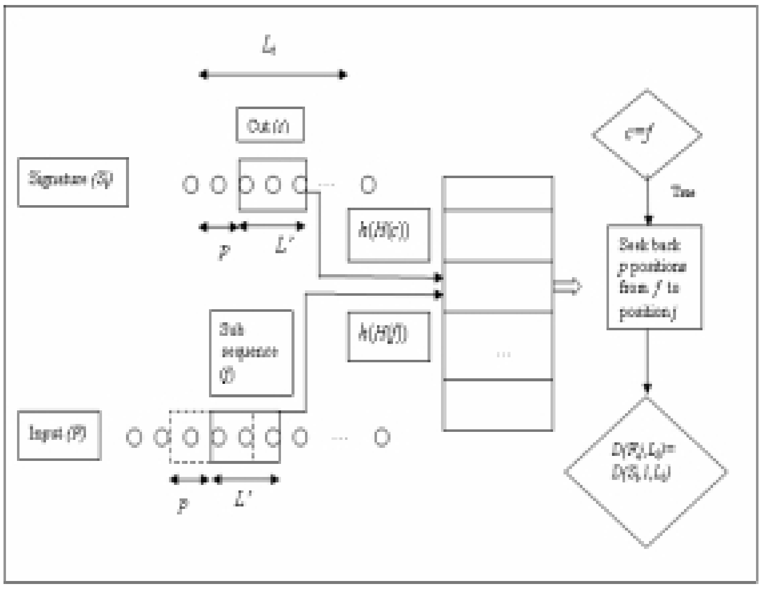


Fig. 1. Illustration of the matching algorithm.

usage. However, since it is used to filter out mismatches, in order to achieve fast scanning speed, it must be long enough such that it does not often match with cuts of non malicious file subsequences. Hence, we choose the cut length as the minimum length where the average of all the false alarm rates for all signature cuts of that length is lower than a fixed threshold.

Let $S_{ij}^{L'}$ represent a cut from signature S_i starting from position j of length L', we choose the minimum L' such that:

$$\frac{1}{N}\sum_{i=1}^{N}\left(\frac{1}{Li-L'+1}\right)\sum_{j=1}^{Li-L'+1} P(S_{ij}^{L'}) < \mu, \tag{16}$$

where $P(S_{ij}^{L'})$ is the false alarm rate of the cut and μ is a threshold factor. We describe the selection of the threshold factor and cut length in Section 5.3.

4. Illustration of the signature matching algorithm

Figure 1 shows an illustration of the signature matching algorithm. The first step is to build a hash table to store the signature information. For this, we choose a signature cut set from the signature set as described in Section 3.2. For any signature S_i with chosen signature cut c of length L', the filter-hash value $H(c)$ is computed using Eq. (1). The corresponding hash index $h(H(c))$ is computed using Eq. (2). At this index in the hash table, the following information is stored:

a) The signature cut, c
b) The position of the signature cut, p
c) The length of the signature, L_i
d) The DJB hash for the entire signature i.e. $D(S_i, 1, L_i)$ computed using Eq. (8)

For any input file F, the filter-hash value is computed for a subsequence f of length L', using the recursive relation given in Eq. (6). The filter-hash value is used to determine the index of a signature in the hash table using Eq. (2). The subsequence f needs to be matched with this signature. If the computed index corresponds to an empty entry in the hash table, no further processing is required and we proceed to the next subsequence. If an entry corresponding to signature S_i exists at this position, the matching is done as follows:

Step 1: The signature cut c is matched against the subsequence f byte by byte. In the case of a positive match, we proceed to perform step 2. In the case of a mismatch we go to next subsequence in the file.

Step 2: Seek back p positions from the start of subsequence f in the input file, where p is the cut position for S_i.

Step 3: Compute the DJB hash starting from the current position in the input for a length L_i using Eq. (8).

Step 4: Compare the stored DJB hash of the signature with computed DJB hash. An exact match indicates malware detection.

5. Performance analysis

5.1. Memory

Mobile phone applications have to work with stringent memory requirements. The hashed representation needs less memory for storing signature information. This is because we store DJB hash values of signatures instead of the signatures themselves. In addition, we store the signature cut and the cut position in the hash table. The signature cut length L' is chosen to be much lesser than the signature length and is constant for all signatures. Memory required for one signature $= L' + 4 + 1 + 1$, since the DJB hash can be stored in 4 bytes, the cut position can be stored in 1 byte and the length of the signature can be stored in 1 byte. We choose a cut length of size 4 for our implementation. This means, we use a total of 10 bytes for each signature and $10 * N$ bytes for N signatures. In general, every signature requires constant memory. This is an improvement from matching algorithms that suggest storing the actual signatures themselves in memory. In such cases the average memory consumption for N signatures is $O(N * L_{avg})$, where L_{avg} is the average length of a signature in bytes. The size of the signature is generally large enough to uniquely identify the virus with a very low probability of false alarms. Also, with every new malware potentially requiring a new signature, we expect the memory savings to be significant as the number of malware continue to grow.

5.2. Time complexity

The scanning method pre-computes and stores the DJB hash value and signature cut for a signature at the index computed by the signature cut's filter-hash value. For an input subsequence, the average time complexity for matching this subsequence with a signature in the hash table is given by $O(\zeta + \lambda * L_x)$, where ζ is the average number of comparisons required to match the signature cut with the input subsequence, λ is the mismatch probability which determines if the DJB hash needs to be computed for a given subsequence and L_x is the average length of a signature. The DJB hash needs to be computed every time the signature cut matches the subsequence. Hence the mismatch probability is the probability that

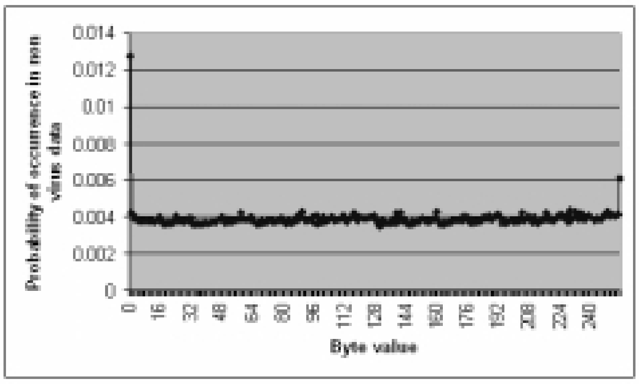


Fig. 2. Occurrence of bytes in non malicious data.

the signature cut exactly matches an input subsequence but the DJB hash value of the complete signature and input subsequence do not match. The above equation means that the best case running time of the algorithm is when λ reduces to 0 i.e. no mismatches occur. In this case the total running time reduces to $O(\zeta)$. We show that λ has a low value using experimental results. The average number of comparisons to match the signature cut $S_{ij}^{L'} = (s_j s_{j+1} \ldots s_{j+L'})$ and the input subsequence $F' = (f_j f_{j+1} \ldots f_{j+L'})$ of length L' is given by:

$$\zeta = 1 + \sum_{k=j+1}^{j+L'} P(sk = fk | S_{ik-1}^{L'} = F'k - 1), \tag{17}$$

where $S_{ik-1}^{L'} = F'_{k-1}$ indicates that bytes j to $j+k-1$ in $S_{ij}^{L'}$ and F' are identical. Since, we minimize the false alarm rate of the signature cut, we expect $P(s_k = f_k)$ to be sufficiently low for any non malicious file which would reduce the average number of comparisons required to match the signature cut with the input subsequence.

5.3. Results

Symbian OS is the most popular mobile platform for smart phones. It is also the platform that has been targeted by virus writers and has the most number of mobile viruses [28]. Hence, we perform the testing and evaluation of our matching algorithm on Symbian OS devices.

Currently we have about 82 virus signatures with the shortest length equal to 32 bytes and the longest equal to 128 bytes. These signatures are capable of detecting 362 mobile malware, which constitute most of the current mobile viruses. In order to estimate the probabilities of occurrence, we use statistics obtained out of a set of 1000 non malicious mobile applications.

Our optimization of the signature matching process is based on the fact that some data bytes occur more frequently in non malicious data than others. This is illustrated by Fig. 2 which shows the probability of occurrence for every possible single byte value between 0 and 255. These values are obtained by collecting the frequency of occurrence for every possible byte values in the non malicious training data. As we can observe, bytes like 0 are more dominant in non malicious data than the other bytes. We use this knowledge to minimize the number of comparisons during signature matching.

The relevance between bytes measured by Eq. (11) based on their mutual information is illustrated in Fig. 3. It is observed that the maximum value of mutual information is obtained when the separation between bytes is equal to 1. Hence, $l = 1$ is used in the computation of false alarm rates using Eq. (13).

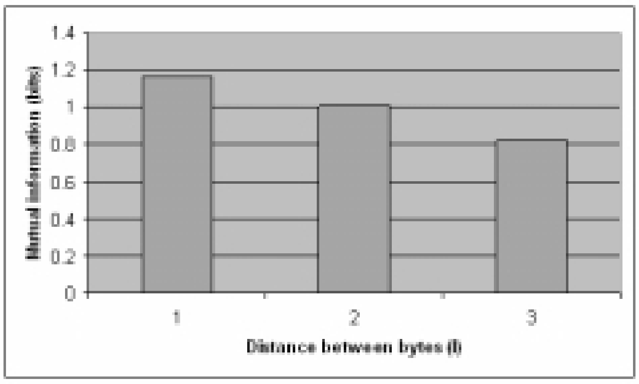


Fig. 3. Mutual information between bytes.

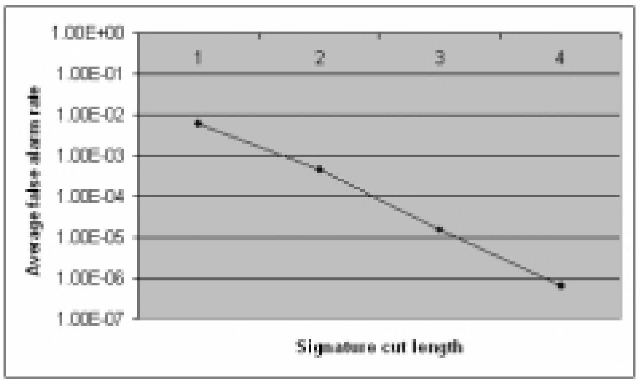


Fig. 4. Drop in average false alarm rate with increasing length.

Figure 4 illustrates the drop in false alarm rates as the signature cut size increases. It is observed that the average false alarm rate drops by nearly 100% with a single byte increase in signature cut length. We need to have a balance between cut length and this probability i.e. we do not want to store a larger cut length but at the same time maintain a low average false alarm rate for that length. We choose a value of 10^{-6} for the threshold factor, μ in Eq. (16) to choose the signature cut length. Hence, using our results, we use a cut length of size 4 bytes to compute the signature cuts.

As mentioned in Section 2.1, the size of a hash table can play a significant role in determining the scanning speed. Figure 5 shows the variation of scanning speed based on the size of the hash table for varying number of signatures. We can see that initially the scanning-time/Kilobyte of data decreases rapidly since there are more empty slots in the hash table. But beyond a certain size, the scanning time remains steady or increases due to increased time for memory access in a large table. It should be noted here that since there are very few mobile malware signatures at the moment, in order to evaluate our solution for larger signature sets and consequently larger hash table sizes, we use random sequences of data ranging between 16–64 bytes as signatures while obtaining the results.

Using a hash table reduces scanning time significantly when compared to sequential matching. The comparison of sequential scanning against scanning using a hash table gives the following result: scanning using a hash table has a scanning speed of 0.037 millisecs/KB while scanning using sequential scanning takes 2.034 millisecs/KB of data. This means that scanning using a hash table is faster than sequential

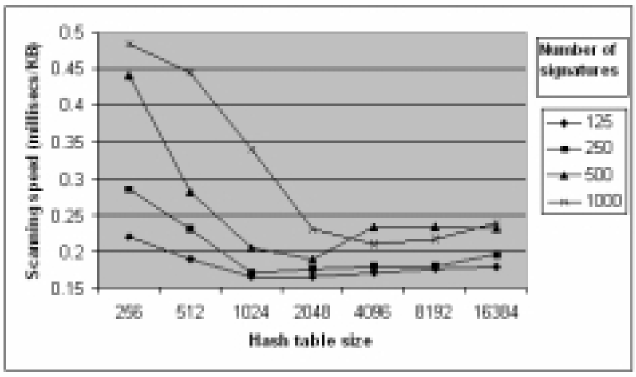


Fig. 5. Estimating optimal hash table size using scanning speed.

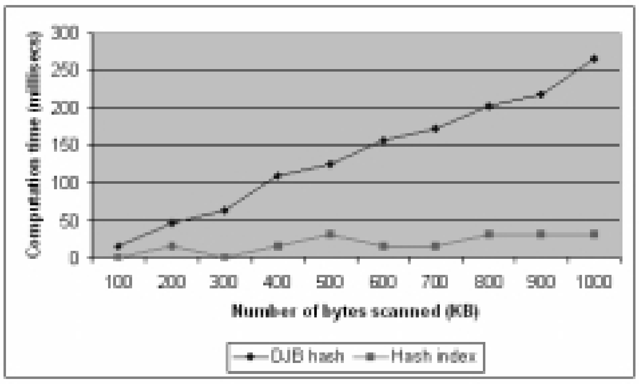


Fig. 6. Comparison of speed of DJB hash computation with speed of hash index computation.

scanning by more than 98%.

The filter-hash and hash index computation for the signature cut using Eqs (6) and (2) should be computationally efficient. This is necessary since this computation is repeated for every input subsequence. In contrast, the DJB hash computation is only necessary for a very small percentage of input subsequences. Figure 6 shows the comparison of speeds of DJB hash computations with hash index computations using Eqs (6) and (2). As we can see the hash index computation speed remains fairly steady compared to the growth of the DJB hash computation speed with input size. The mismatch probability i.e. matching the filter-hash but not the DJB hash plays an important part in determining the running time of our scanning method. This controls the number of times the DJB hash is computed for an input file. Figure 7 illustrates the variance of the mismatch probability for random input bytes. On average the mismatch probability is about 10^{-6}. This means that the computationally expensive DJB hash value needs to be computed a negligible number of times for non malicious input data which results in speeding up the scanning algorithm.

Figure 8 shows the comparison of our scanning method i.e. signature cut matching (and DJB hash matching if necessary), against conventional matching. In conventional matching we store the entire signature and start matching from the first byte whenever the hash index points to a non empty slot in the hash table. The results suggest that our method can speed up scanning as the number of signatures increase. Our results show that on average our scanning method takes 0.0013 millisecs/KB/signature

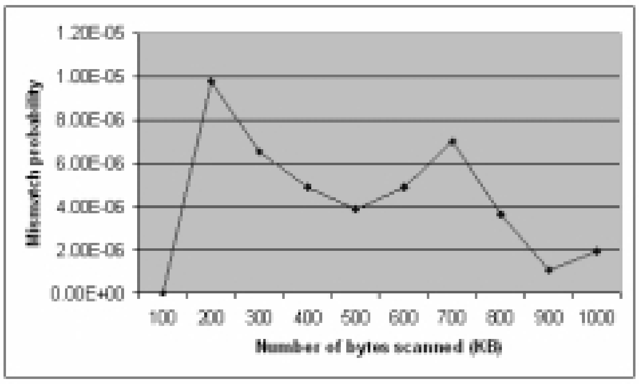


Fig. 7. Estimating mismatch probability.

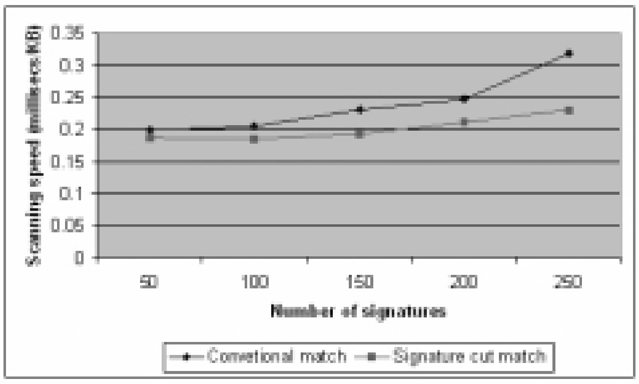


Fig. 8. Comparison of scanning speed using conventional matching against using signature cut matching for varying number of signatures.

while the conventional matching takes 0.0016 millisecs/KB/signature. The difference may also be due to the extra memory consumed for storage of signatures in the hash table which can potentially increase access time. It should be noted that for the purpose of this comparison, we have implemented a naïve matching algorithm that starts matching from the first byte till it finds a mismatch. Better matching methods as mentioned in section 1 may provide different results.

5.4. *Comparison with Clam-AV scanner*

Clam-AV is the best known and most widely used open source virus scanner [31]. It uses signature based virus detection. In particular, it uses the well-known Aho-Corasick multi pattern matching algorithm. Hence, we compare our solution against the Clam-AV scanner to test its effectiveness. We adapted the Clam-AV scanner to run on the mobile platform for testing purposes and compared it with our solution.

5.4.1. *Accuracy of detection*

Our scanner was allowed to run on multiple handsets with different system files and different third party application software. During many months of testing, our software has not given any false alarms.

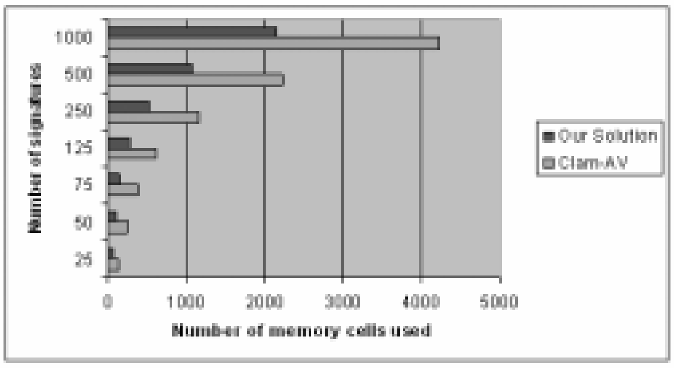


Fig. 9. Comparison of memory usage of Clam-AV with our solution.

It is capable of detecting all the current mobile virus samples i.e. 362 of them available to us thus far. In contrast, even though Clam-AV did not give any false alarms, even with its updated signature set it could only detect 232 of our virus samples i.e. it missed about 35% of mobile viruses.

5.4.2. Computational efficiency

The Clam-AV signature set consists of definitions for PC as well as mobile viruses which makes it a large signature DB. In order to make the comparison fair, we use a common signature set for both our solution as well as clam-AV. It should be noted here that Clam-AV also handles regular expression signatures which contain wildcard characters like "*" and "?". Since we do not currently have sophisticated mobile viruses for e.g. polymorphic viruses that require regular expression matching, we do not have such signatures in our set and have not used it for our testing. However our multi-pattern matching algorithm is extensible so as to incorporate wildcard characters in a signature. In this case, we consider the signature as a multi-part signature where each part is separated by one of the two wildcard characters. Each part of the signature is treated as a single signature by itself and stored at a separate location in the hash table described in Section 2.1. A detection of a multi-part signature requires the successful detection of all its constituent parts.

Since the number of mobile malware signatures is small at the moment, we have utilized random sequences of data ranging between 16 to 64 bytes as signatures in order to evaluate the performance of our solution for a large signature set. In terms of memory, our solution performs much better than the Clam-AV scanner. Figure 9 illustrates the comparison of our scanner with Clam-AV in terms of memory. We use a Symbian built-in function to compute the number of memory cells allocated to test the memory usage. According to our results, our solution uses less than half the amount of memory used by Clam-AV during matching. Specifically, our solution requires on average 2.317778 memory cells/ signature whereas the Clam-AV scanner uses up 5.368889 memory cells/signature. The memory savings will be more pronounced as the number of mobile malware and corresponding signatures increase. In terms of scanning speed, our solution performs as well as Clam-AV. Figure 10 illustrates the comparison of scanning times. As we see from our results our solution performs as good as and sometimes better than Clam-AV. According to our results, our solution has a scanning speed of 0.168643 millisecs/Kilobyte whereas the Clam-AV scanner's speed is 0.17906 millisecs/Kilobyte for the current number of malware signatures.

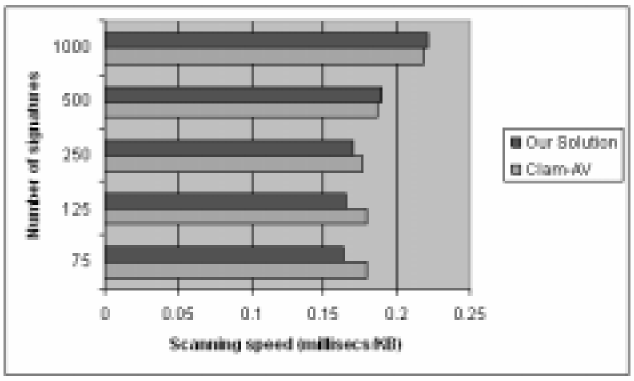


Fig. 10. Comparison of scanning speed of Clam-AV with our solution.

6. Related work

Signature based methods have been used for PC virus scanners and network level intrusion detection systems. As mentioned before, Clam-AV is a well known open source anti virus system for PC domain that uses signature based scanning. Some additions to Clam-AV have been suggested as well [7,11]. Miretskiy et al. [7] have added on-access functionality to the Clam-AV scanner. Attig et al. [4] and Erdogan et al. [11] have suggested the use of bloom filters for scanning and intrusion detection systems at the network level. Katz et al. have suggested a fast pattern matching scheme for high data rate network packet inspection [12]. Fisk et al. have described a set-wise Boyer Moore pattern matching algorithm used in their intrusion detection system [13]. Snort is one of the most well known network level intrusion detection systems whose pattern matching engine is also based on the Boyer-Moore algorithm [20]. Zhang et al [36] have suggested a general framework for intrusion detection on wireless networks. Recently, approaches to intrusion detection specific to mobile ad-hoc networks have been suggested as well [5]. However, the intrusion detection systems are used for network level packet filtering and hence have access to higher computational and memory resources unlike mobile devices and hence have design criteria different from those for mobile applications. Since the occurrence of mobile malware is fairly recent, we could not find work specific to signature based malware detection on mobile devices with which we could compare our solution.

7. Conclusion and future work

The impact of mobile devices and mobile malware on our daily lives cannot be underestimated. It is essential to give due attention to the computational limitations of mobile devices in order to design an effective solution. In this paper, we have described signature based malware detection that is well suited for use in mobile devices. In particular, we have described a signature matching method with low memory usage, fast scanning speed and also provided detailed results of its performance on a handset. The results have shown that our solution performs well when compared to the Clam-AV scanner and provides huge memory savings while maintaining fast scanning speed.

As mentioned earlier, sophisticated techniques like heuristic detection may be explored to detect unknown viruses. Future work may involve developing heuristic detection techniques in addition to signature based detection. An important issue for the mobile domain is to make the heuristic detection computationally efficient. This approach can be explored as the number of mobile malware increase and it becomes possible to obtain stable virus features using a large number of virus samples.

Acknowledgment

The authors would like to thank SMobile Systems Inc. for their support in this work.

References

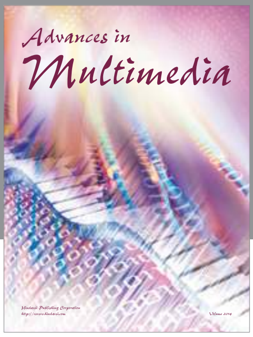
[1] A.V. Aho and M.J. Corasick, Efficient string matching: An aid to bibliographic search, *Communications of the ACM* **18**(6) (1975), 333–340.
[2] Amir Herzberg, Payments and banking with mobile personal devices, *Communications of the ACM* **46**(5) (2003), 53–58.
[3] Arash Partow, DJB Hash function efficiency, URL http://www.vak.ru/doku.php-/proj/hash/efficiency-en.
[4] M. Attig, S. Dharmapurikar and J. Lockwood, Implementation results of bloom filters for string matching, in: *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004, 322–323.
[5] B.M. Reshmi and S.S. Manvi, Bhagyavati, An agent based intrusion detection model for mobile ad hoc networks, *Mobile Information Systems* **2**(4) (2006), 169–191, IOS Press.
[6] B. Commentz-Walter, A string matching algorithm fast on the average, in: *Proceedings of the 6th International Colloquium on Automata, Languages and Programming*. LNCS, vol. 71, Springer-Verlag, Berlin, 1979, 118–132.
[7] C.P.W.Y. Miretskiy, A. Das and E. Zadok, Avfs: An on-access anti-virus file system, in: *Proceedings of the 13th USENIX Security Symposium*, 2004, 73–88.
[8] D. Jaggar, ARM Architecture and Systems, *IEEE Micro* **17**(4) (1997), 9–11.
[9] D. Dagon, T. Martin and T. Starner, Mobile phones as computing devices: the viruses are coming!, *Pervasive Computing, IEEE* **3** (2004), 11–15.
[10] D.E. Knuth, J.H. Morris and V.R. Pratt, Fast Pattern Matching in Strings, *SIAM Journal on Computing* **6**(2) (1977), 323–350.
[11] Erdogan, O. Pei Cao, Hash-AV: Fast Virus Signature Scanning by Cache-Resident Filters, in: *IEEE Global Telecommunications Conference*, 2005, 1767–1772.
[12] Fang Yu Katz, R.H. Lakshman, T.V., Gigabit rate packet pattern-matching using TCAM, in: *Proceedings of the 12th IEEE International Conference on Network Protocols*, 2004, 174–183.
[13] M. Fisk and G. Varghese, *Fast content-based packet handling for intrusion detection,* Tech. Rep. CS2001-0670, University of California, San Diego, 2001.
[14] G. Tesauro, J.O. Kephart and G.B. Sorken, Neural Networks for Computer Virus Recognition, *IEEE Expert* **11**(4) (1996), 5–6.
[15] J. Jamaluddin, N. Zotou and P. Coulton, Mobile phone vulnerabilities: a new generation of malware, in: *IEEE International Symposium on Consumer Electronics*, 2004, 199–202.
[16] J.O. Kephart and W.C. Arnold, Automatic extraction of Computer Virus Signatures, in: *4th Virus Bulletin International Conference*, 1994, 178–184.
[17] J.O. Kephart, G.B. Sorkin, M. Swimmer and S.R. White, Blueprint for a Computer Immune System, *Artificial Immune Systems and Their Applications*, Springer-Verlag, 1999, 221–241.
[18] D.N. Knisely, S. Kumar, S. Laha and S. Nanda, Evolution of wireless data services: IS-95 to cdma2000, *Communications Magazine, IEEE* **36**(10) (1998), 140–149.
[19] M.G. Schultz, E. Eskin, E. Zadok and S.J. Stolfo, Data Mining Methods for Detection of New Malicious Executables, in: *IEEE Symposium on Security and Privacy*, 2001, 38–49.
[20] M. Roesch, Snort – Lightweight Intrusion Detection for Networks, in: *Proceedings of the 13th LISA Conference*, 1999, 229–238.
[21] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W.H Freeman & Co., San Francisco, 1979.
[22] S. Ohmori, Y. Yamao and N. Nakajima, The future generations of mobile communications based on broadband access technologies, *Communications Magazine, IEEE* **38**(12) (2000), 134–142.
[23] P. Szor, *The art of computer Virus Research and Defense,* Symantec Press, 2005, 225–295.
[24] R.S. Engelschall, on DJB hash. URL http://gcov.php.net/PHP_5_1/lcov/php-src/Zend/zend_hash.h.gcov.php.
[25] R. Harrison et al., *Symbian OS C++ for mobile phones,* Wiley, 2003, 22–25.
[26] R.M. Karp and M.O. Rabin, Efficient randomized pattern-matching algorithms, *IBM Journal of Research and Development* **31**(2) (1987), 249–260.

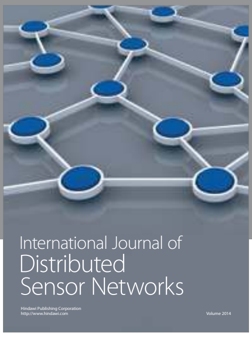
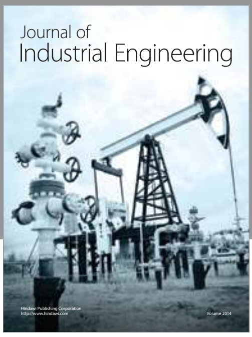
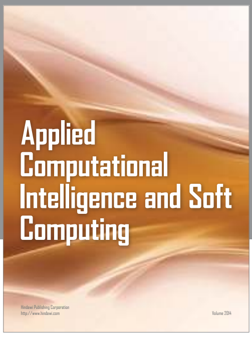
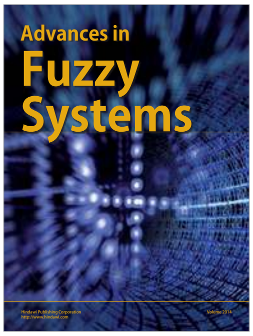
[27] R.S. Boyer and J. Strother Moore, A fast string searching algorithm, *Communications of the ACM* **20**(10) (1977), 762–772.
[28] R.X. Wang, *Symbian OS–mysterious playground for new malware,* Virus Bulletin, September 2005.
[29] S. Sahni, Approximate algorithms for 0/1 knapsack problem, *Journal of the ACM* **22**(11) (1975), 115–124.
[30] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction To Algorithms,* McGraw-Hill, 2002, 966–1057.
[31] T. Kojm, *Clamav anti-virus,* URL http: //www.clamav.net/.
[32] T.M. Cover and J.A. Thomas, *Elements of Information Theory,* New York: Wiley, 1991.
[33] D. Venugopal, An efficient signature representation and matching method for mobile devices, in: *Proceedings of Conference on Wireless Internet*, 2006.
[34] W.D. Maurer and T.G. Lewis, Hash table methods, *ACM Computing Surveys* **7** (1975), 5–19.
[35] S. Wu and U. Manber, Agrep – a fast approximate pattern-matching tool, in: *Proceedings of the Usenix Winter 1992 Technical Conference*, 1992, 153–162.
[36] Y. Zhang, W. Lee and Yi-an Huang, Intrusion Detection Techniques for Mobile Wireless Networks, *Wireless Networks* **9**(5) (2003), 545–556.

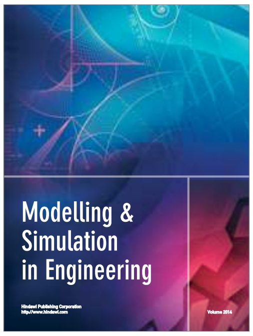
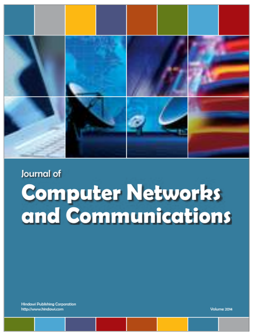
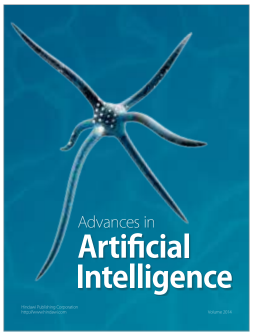
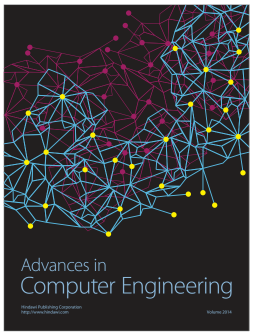
Deepak Venugopal obtained his Bachelor's degree in Information Science and Engineering from Bangalore University in 2001 and Master's degree in Computer Science from The University of Texas at Dallas in 2004. He was with SMobile Systems Inc. from 2005 to 2007. He currently works as a Senior Software Engineer at Nokia Inc. His main research interests are in the areas of security on mobile devices/networks and algorithms for constrained devices.

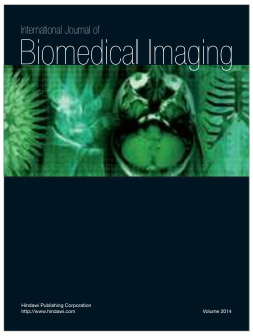
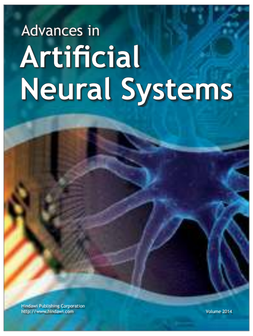
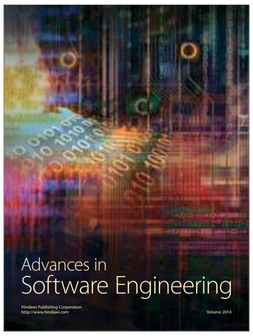
Guoning Hu received the B.S. and M.S. degrees in physics from Nanjing University, Nanjing, China, in 1996 and 1999, respectively, and the Ph.D. degree in Biophysics from the Ohio State University in 2006.

From October 2005 to May 2006, he was with SMobile Systems Inc., Columbus, Ohio. He is currently a Senior Software Engineer at Truveo Video Search, AOL. Dr. Hu's research interests include malware detection, mobile security, and statistical machine learning.

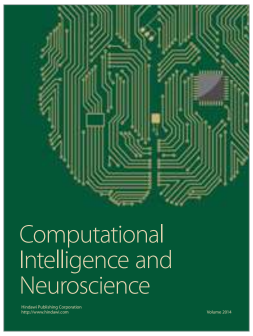

Advances in Multimedia

The Scientific World Journal

International Journal of Distributed Sensor Networks

Journal of Industrial Engineering

Applied Computational Intelligence and Soft Computing

Advances in Fuzzy Systems


Modelling & Simulation in Engineering

Journal of Computer Networks and Communications

Advances in Artificial Intelligence

Advances in Computer Engineering

International Journal of Computer Games Technology

International Journal of Biomedical Imaging

Advances in Artificial Neural Systems

Advances in Software Engineering

Journal of Robotics

Advances in Human-Computer Interaction

Computational Intelligence and Neuroscience

International Journal of Reconfigurable Computing

Journal of Electrical and Computer Engineering