Inf Syst Front (2008) 10:33–45
DOI 10.1007/s10796-007-9054-3

# A scalable multi-level feature extraction technique to detect malicious executables

**Mohammad M. Masud · Latifur Khan · Bhavani Thuraisingham**

Published online: 23 October 2007


**Abstract** We present a scalable and multi-level feature extraction technique to detect malicious executables. We propose a novel combination of three different kinds of features at different levels of abstraction. These are binary *n*-grams, assembly instruction sequences, and Dynamic Link Library (DLL) function calls; extracted from binary executables, disassembled executables, and executable headers, respectively. We also propose an efficient and scalable feature extraction technique, and apply this technique on a large corpus of real benign and malicious executables. The above mentioned features are extracted from the corpus data and a classifier is trained, which achieves high accuracy and low false positive rate in detecting malicious executables. Our approach is knowledge-based because of several reasons. First, we apply the knowledge obtained from the binary *n*-gram features to extract assembly instruction sequences using our Assembly Feature Retrieval algorithm. Second, we apply the statistical knowledge obtained during feature extraction to select the best features, and to build a classification model. Our model is compared against other feature-based approaches for malicious code detection, and found to be more efficient in terms of detection accuracy and false alarm rate.

**Keywords** Disassembly · Feature extraction · Malicious executable · *n*-gram analysis

M. M. Masud
Department of Computer Science,
The University of Texas at Dallas,
2700 Waterview Pkwy, #5116,
Richardson, TX 75080, USA
e-mail: mehedy@utdallas.edu

L. Khan · B. Thuraisingham (✉)
Department of Computer Science,
The University of Texas at Dallas,
Box 830688, EC 31,
Richardson, TX 75083-0688, USA
e-mail: bhavani.thuraisingham@utdallas.edu

L. Khan
e-mail: lkhan@utdallas.edu

## 1 Introduction

Malicious code is a great threat to computers and computer society. Numerous kinds of malicious codes wander in the wild. Some of them are mobile, such as worms, and spread through internet causing damage to millions of computers worldwide. Other kinds of malicious codes are static, such as viruses, but sometimes deadlier than its mobile counterpart. Malicious code writers usually exploit software vulnerabilities to attack host machines. A number of techniques have been devised by researchers to counter these attacks. Unfortunately, the more successful the researchers become in detecting and preventing the attacks, the more sophisticated malicious code appears in the wild. Thus, the battle between malicious code writers and researchers is virtually never-ending.

One popular technique followed by the anti-virus community to detect malicious code is "signature detection." This technique matches the executables against a unique telltale string or byte pattern called *signature*, which is used as an identifier for a particular malicious code. Although signature detection techniques are being used widely, they are not effective against *zero-day* attacks (new

malicious code), *polymorphic* attacks (different encryptions of the same binary), or *metamorphic* attacks (different code for the same functionality). So, there has been a growing need for fast, automated, and efficient detection techniques that are robust to these attacks. As a result, many automated systems (Newsome et al. 2005; Kolter and Maloof 2004; Golbeck and Hendler 2004; Newman et al. 2002) have been developed.

In this paper we describe our novel *hybrid feature retrieval* (HFR) model that can detect malicious executables efficiently. This is an extension to our previous work (Masud et al. 2007b). It extracts three different kinds of features from the executables at different levels of abstraction and combines them into one feature set, called the *hybrid feature set* (HFS). These features are used to train a classifier (e.g. *support vector machine* (SVM), *decision tree* etc.), which is applied to detect malicious executables. These features are: (a) binary *n*-gram features, (b) derived assembly features (DAFs), and (c) dynamic link library (DLL) call features. Each binary *n*-gram feature is actually a sequence of *n* consecutive bytes in a binary executable, extracted using a technique explained in Section 3.1. Binary *n*-grams reveal the distinguishing byte patterns between the benign and malicious executables. Each DAF is a sequence of assembly instructions in an executable, and corresponds to one binary *n*-gram feature. DAFs reveal the distinctive instruction usage patterns between the benign and malicious executables. They are extracted from the disassembled executables using our *assembly feature retrieval* (AFR) algorithm, explained in Section 4.2. It should be noted that DAF is different from assembly *n*-gram features mentioned in Section 3.2. Assembly *n*-gram features are not used in HFS because of our findings that DAF performs better than them. Each DLL call feature actually corresponds to a DLL function call in an executable, extracted from the executable header as explained in Section 3.3. These features reveal the distinguishing DLL call patterns between the benign and malicious executables. We show empirically that the combination of these three features is always better than any single feature in terms of classification accuracy.

Our work focuses on expanding features at different levels of abstraction, rather than using more features at a single level of abstraction. There are two main reasons behind this. First, the number of features at a given level of abstraction (e.g. binary) is overwhelmingly large. For example, in our larger dataset, we obtain 200 million binary *n*-gram features. Training with this large number of features is way beyond the capabilities of any practical classifier. That is why we limit the number of features at a given level of abstraction to an applicable range. Second, we empirically observe the benefit of adding more levels of abstraction to the combined feature set (i.e., HFS). HFS combines features at three levels of abstraction, namely, binary executables, assembly programs, and system API calls. We show that this combination has higher detection accuracy and lower false alarm rate than the features at any single level of abstraction.

Our technique is related to knowledge-management because of several reasons. First, we apply our knowledge of binary *n*-gram features to obtain DAFs. Second, we apply the knowledge obtained from the feature extraction process to select the best features. This is accomplished by extracting all possible binary *n*-grams from the training data, applying the statistical knowledge corresponding to each *n*-gram (i.e., its frequency in malicious and benign executables) to compute its *information gain* (Mitchell 1997), and selecting the best *S* of them. Finally, we apply another statistical knowledge (presence/absence of a feature in an executable) obtained from the feature-extraction process to train classifiers.

Our research contributions are as follows. First, we propose and implement our HFR model, which combines three kinds of features mentioned above. Second, we apply a novel idea to extract assembly instruction features using binary *n*-gram features, implemented with the AFR algorithm. Third, we propose and implement a scalable solution to the *n*-gram feature extraction and selection problem in general. Our solution works well with limited memory, and significantly reduces running time by applying efficient and powerful data structures and algorithms. Thus, it is scalable to large collection of executables (in the order of thousands), even with limited main memory and processor speed. Finally, we compare our results against Kolter & Maloof's results (Kolter and Maloof 2004), which uses only binary *n*-gram feature, and show that our method achieves better accuracy. We also report the performance/cost tradeoff of our method against Kolter & Maloof's method. It should be pointed out here that our main contribution is an efficient feature extraction technique, not a classification technique. We empirically prove that the combined feature set (i.e., HFS) extracted using our algorithm performs better than other individual feature sets (such as binary *n*-grams) regardless of the classifier (e.g. SVM / decision tree) used.

The rest of the paper is organized as follows: Section 2 discusses related work, Section 3 presents and explains different kinds of *n*-gram feature extraction techniques, Section 4 describes the HFR model, Section 5 discusses our experiments and analyzes results, Section 6 concludes with future research directions.

## 2 Related work

There have been significant research works in recent years to detect malicious executables. There are two mainstream techniques to automate the detection process: behavioral and content-based. The behavioral approach is primarily applied to detect mobile malicious code. This technique is applied to analyze network traffic characteristics such as source-destination ports/IP addresses, various packet level / flow level statistics, and application level characteristics such as email attachment type, attachment size etc. Examples of behavioral approaches include social network analysis (Golbeck and Hendler 2004; Newman et al. 2002) and statistical analysis (Schultz et al. 2001a). A data mining based behavioral approach for detecting email worms has been proposed by Masud et al. (2007a). Garg et al. (2006) apply feature-extraction technique along with machine learning for *masquerade detection*. They extract features from user behavior in GUI-based systems, such as mouse speed, number of clicks per session and so on. Then the problem is modeled as a binary classification problem, and trained and tested with SVM. Our approach is content-based, rather than behavioral.

The content-based approach analyzes the content of the executable. Some of them try to automatically generate signatures from network packet payloads. Examples are EarlyBird (Singh et al. 2003), Autograph (Kim and Karp 2004), and Polygraph (Newsome et al. 2005). In contrast, our method does not require signature generation or signature matching. Some other content-based techniques extract features from the executables and apply machine learning to detect malicious executables. Examples are Schultz et al. (2001b) and Kolter and Maloof (2004). Schultz et al. (2001b) extract DLL call information using GNU Bin-Utils (Cygnus 1999) and character strings using GNU *strings*, from the header of Windows PE executables. Also, they use byte sequences as features. We also use byte sequences and DLL call information, but we also apply disassembly and use assembly instructions as features. Besides, we extract byte patterns of various lengths (from 2 to 10 bytes), whereas they extract only 2-byte length patterns. A similar work is done by Kolter et al. (Kolter and Maloof 2004). They extract binary *n*-gram features from the binary executables and apply them to different classification methods, and report accuracy. Our model is different from (Kolter and Maloof 2004) in that we extract not only the binary *n*-grams but also assembly instruction sequences from the disassembled executables, and gather DLL call information from the program headers. We compare our model's performance only with (Kolter and Maloof 2004), since they report higher accuracy than (Schultz et al. 2001b).

## 3 Feature extraction using n-gram analysis

Before going into details of the process, we illustrate a code snippet in Fig. 1 from the Email-Worm "Win32.Ainjo.e," and use it as a running example throughout the paper.

Feature extraction using *n*-gram analysis involves extracting all possible *n*-grams from the given dataset (*training set*), and selecting the best *n*-grams among them. Each such *n*-gram is a feature. We extend the notion of *n*-gram from bytes to assembly instructions, and DLL function calls. That is, an *n*-gram may be either a sequence of *n* bytes, *n* assembly instructions, or *n* DLL function calls, depending on whether we are to extract features from binary executables, assembly programs, or DLL call sequences, respectively. Before extracting *n*-grams, we preprocess the binary executables by converting them to *hexdump* files and *assembly program* files, as explained shortly.

### 3.1 Binary *n*-gram feature

Here the granularity level is a *byte*. We apply the UNIX 'hexdump' utility to convert the binary executable files into text files, mentioned henceforth as 'hexdump files,' containing the hexadecimal numbers corresponding to each byte of the binary. This process is performed to ensure safe and easy portability of the binary executables. The feature extraction process consists of two phases: (a) feature collection, and (b) feature selection, both of which are explained in the following subsections.

```
------------------------------------------------------------
CODE SNIPPET:-

Program Entry Point = 00472E70
(Email-Worm.Win32.Ainjo.e File offset:00000400)

 address  opcode                  assembly
--------- ------                  ------------------------
:00455000 FF21                    jmp dword[ecx]
:00455002 089000270014            or byte[eax+14002700], dl
:00455008 00761E                  add byte[esi+1E], dh
:0045500B 45                      inc ebp
:0045500C 00DE                    add dh, bl
:00455010 B4DE                    mov ah, -22

------------------------------------------------------------
DLL FUNCTION CALL INFO FROM THE HEADER

   Module : KERNEL32.DLL

Addr:00073CAE Name: LoadLibraryA
Addr:00073CBC Name: GetProcAddress
Addr:00073CCC Name: ExitProcess
------------------------------------------------------------
```

**Fig. 1** Code snippet and DLL call info from the email-worm "Win32.Ainjo.e"

### 3.1.1 Feature collection

We collect binary *n*-grams from the 'hexdump' files. This is illustrated in example-I.

---

Example-I

The 4-grams corresponding to the first 6 bytes sequence (FF2108900027) from the executable in Figure 1 are the 4-byte sliding windows: FF210890, 21089000, and 08900027

---

The basic feature collection process runs as follows. At first, we initialize a list $L$ of *n*-grams to empty. Then we scan each hexdump file by sliding an *n*-byte window. Each such *n*-byte sequence is an *n*-gram. Each *n*-gram $g$ is associated with two values: $p_1$ and $n_1$, denoting the total number of positive instances (i.e., malicious executables) and negative instances (i.e., benign executables), respectively, that contain $g$. If $g$ is not found in $L$, then $g$ is added to $L$, and $p_1$ and $n_1$ are updated as necessary. If $g$ is already in $L$, then only $p_1$ and $n_1$ are updated. When all hexdump files have been scanned, $L$ contains all the unique *n*-grams in the dataset along with their frequencies in the positive and negative instances. There are several implementation issues related to this basic approach. First, the total number of *n*-grams may be very large. For example, the total number of 10-g in our second dataset (see Section 5.1) is 200 million. It may not possible to store all of them in computer's main memory. To solve this problem, we store the *n*-grams in a disk file $F$. Second, if $L$ is not sorted, then a linear search is required for each scanned *n*-gram to test whether it is already in $L$. If $N$ is the total number of *n*-grams in the dataset, then the time for collecting all the *n*-grams would be $O(N^2)$, an impractical amount of time when $N$=200 million.

In order to solve the second problem, we use a data structure called Adelson Velsky Landis (AVL) tree (GoodRich and Tamassia 2006) to store the *n*-grams in memory. An AVL tree is a height-balanced binary search tree. This tree has a property that the absolute difference between the heights of the left sub-tree and the right sub-tree of any node is at most one. If this property is violated during insertion or deletion, a balancing operation is performed, and the tree regains its height-balanced property. It is guaranteed that insertions and deletions are performed in logarithmic time. So, in order to insert an *n*-gram in memory, we now need only $O(\log_2(N))$ searches. Thus, the total running time is reduced to $O(N\log_2(N))$, making the overall running time about 5 million times faster for $N$ as large as 200 million. Our feature collection algorithm *Extract_Feature* implements these two solutions. It is illustrated in Algorithm 1.

Description of the algorithm: the *for* loop at line 3 runs for each hexdump file in the training set. The inner *while* loop at line 4 gathers all the *n*-grams of a file and adds it to the AVL tree if it is not already there. At line 8, a test is performed to see whether the tree size has exceeded the memory limit (a threshold value). If it exceeds and $F$ is empty, then we save the contents of the tree in $F$ (line 9). If $F$ is not empty, then we merge the contents of the tree with $F$ (line 10). Finally, we delete all the nodes from the tree (line 12).

Time Complexity, $T$ = time ($n$ − gram reading & inserting *in tree*) + time (merging with disk) = $O(B\log_2 K) + O(N)$, where $B$ is the total size of the training data in *bytes*, $K$ is the maximum #of nodes of the tree (i.e., threshold), and $N$ is the total number of *n*-grams collected. Space Complexity: O ($K$), where $K$ is defined as above.

**Algorithm 1** The *n*-gram feature collection algorithm

```
Procedure Extract_Feature (B)
B ={ B1 , B2, … , BK} : all hexdump files
1. T ← empty tree // Initialize AVL-tree
2. F ← new file // Initialize disk file
3. for each Bi ∈ B do
4.     while not EOF(Bi) do  //while not end of file
5.         g ← next_ngram(Bi) // read next n-gram
6.         T.insert(g)  // insert into tree and/or update frequencies as necessary
7.     end while
8.     if T.size > Threshold then //save or merge
9.         if F is empty then  F ←T.inorder() //save tree data in sorted order
10.          else F ← merge(T.inorder(), F) //merge tree data with file data and
11.          save
12.        end if
13.        T ← empty tree //release memory
14.    end if
15. end for
```

### 3.1.2 Feature selection

If the total number of extracted features is very large, it may not possible to use all of them for training because of several reasons. First, the memory requirement may be impractical. Second, training may be too slow. Third, a classifier may become confused with a large number of features, because most of them would be noisy, redundant or irrelevant. So, we are to choose a small, relevant and useful subset of features. We choose *information gain* (IG) as the selection criterion, because it is one of the best criteria used in literature for selecting the best features.

IG can be defined as a measure of effectiveness of an attribute (i.e., feature) in classifying a training data (Mitchell 1997). If we split the training data based on the values of this attribute, then IG gives the measurement of the expected reduction in entropy after the split. The more an attribute can reduce entropy in the training data, the

better the attribute is in classifying the data. IG of an attribute $A$ on a collection of instances $I$ is given by Eq. 1:

$$\text{Gain}(I, A) \equiv \text{Entropy}(I) - \sum_{V \in \text{values}(A)} \frac{p_v + n_v}{p + n} \text{Entropy}(I_v) \quad (1)$$

Where

values ($A$) is the set of all possible values for attribute $A$

$I_v$ is the subset of $I$ where all instances have the value of $A = v$

$p$ is the total number of positive instances in $I$

$n$ is the total number of negative instances in $I$

$p_v$ is the total number of positive instances in $I_v$,

$n_v$ is the total number of negative instances in $I_v$

In our case, each attribute has only two possible values, i.e., $v \in \{0, 1\}$. If an attribute $A$ (i.e. an $n$-gram) is present in an instance $X$, then $X_A$=1, otherwise it is 0. Entropy of $I$ is computed using the following Eq. 2:

$$\text{Entropy}(I) = -\frac{p}{p+n} \log_2 \left(\frac{p}{p+n}\right) - \frac{n}{p+n} \log_2 \left(\frac{n}{p+n}\right) \quad (2)$$

Where $I$, $p$, and $n$ are as defined above. Substituting Eq. 2 in Eq. 1 and letting $t = n + p$ we get,

$$\text{Gain}(I, A) \equiv -\frac{p}{t} \log_2 \left(\frac{p}{t}\right) - \frac{n}{t} \log_2 \left(\frac{n}{t}\right) - \sum_{v \in \{0,1\}} \frac{t_v}{t} \left(-\frac{p_v}{t_v} \log_2 \left(\frac{p_v}{t_v}\right) - \frac{n_v}{t_v} \log_2 \left(\frac{n_v}{t_v}\right)\right) \quad (3)$$

Now, the next problem is to select the best $S$ features (i.e., $n$-grams) according to IG. One naïve approach is to sort the $n$-grams in non-increasing order of IG and selecting the top $S$ of them, which requires $O$ ($N\log_2 N$) time and $O$ ($N$) main memory. But this selection can be more efficiently accomplished using a *heap* that requires $O$ ($N\log_2 S$) time and $O(S)$ main memory. For $S$=500 and $N$=200 million, this approach is more than 3 times faster and requires 400,000 times less main memory. A heap is a balanced binary tree with the property that the root of any sub-tree contains the minimum (maximum) element in that sub-tree. We use a *min-heap* that always has the minimum value at its root. Algorithm 2 sketches the feature selection algorithm. At first, the heap is initialized to empty. Then the $n$-grams (along with their frequencies) are read from disk (line 2) and inserted into the heap (line 5) until the heap size becomes $S$. After the heap size becomes equal to $S$, we compare the IG of the next $n$-gram $g$ against the IG of the root. If IG (*root*)=IG ($g$) then $g$ is discarded (line 6) since *root* has the minimum IG. Otherwise, *root* is replaced with $g$ (line 7). Finally, the heap property is *restored* (line 9). The process terminates when there are no more $n$-grams in the disk. After termination, we have the $S$ best $n$-grams in the heap.

**Algorithm 2** The $n$-gram feature selection algorithm

```
Procedure Select_Feature (F, H, p, n)
F: a disk file containing all n-grams
H: empty heap
p: total number of positive examples
n: total number of negative examples
1. while not EOF(F) do
2.   <g, p1, n1> ← next_ngram(F)  //read n-gram with frequency counts
3.   p0 = P-p1, n0 = N- n1 // #of positive and negative examples not containing g
4.   IG ← Gain(p0, n0, p1, n1, p, n) // using equation (3)
5.   if H.size() < S then H.insert(g, IG)
6.      else if IG <= H.root.IG then continue //discard lower gain n-grams
7.      else H.root ← <g, IG> //replace root
8.   end if
9.   H.restore()  //apply restore operation
10. end while
```

The insertion and restoration takes only $O$ ($\log_2(S)$) time. So, the total time required is $O$ ($N\log_2 S$), with only $O(S)$ main memory. We denote the best $S$ binary features selected using IG criterion as the *Binary Feature Set* (BFS).

### 3.2 Assembly $n$-gram feature

In this case, the level of granularity is an assembly instruction. First, we disassemble all the binary files using a disassembly tool called *PEDisassem* (Windows P.E. 1998). It is used to disassemble Windows Portable Executable (P.E.) files. Besides generating the assembly instructions with opcode and address information, PEDisassem provides useful information like list of resources (e.g. cursor) used, list of DLL functions called, list of exported functions, and list of strings inside the code block and so on. In order to extract assembly $n$-gram features, we follow a method similar to the binary $n$-gram feature extraction. First we collect all possible $n$-grams, i.e., sequences of $n$ consecutive assembly instructions, and select the best $S$ of them according to IG. We mention henceforth this selected set of features as *Assembly Feature Set* (AFS). We face the same difficulties as in binary $n$-gram extraction, such as limited memory and slow running time, and solve them in the same way. Example-II illustrates the assembly $n$-gram features.

```
------------------------------------------------
Example-II

The 2-grams corresponding to the first 4
assembly instructions in Figure 1 are the
two-instruction sliding windows:

jmp dword[ecx]            ;  or byte[eax+14002700], dl
or byte[eax+14002700], dl ;  add byte[esi+1E], dh
add byte[esi+1E], dh      ;  inc ebp
------------------------------------------------
```

We adopt a standard representation of assembly instructions that has the following format: *name.param1.param2*. Name is the instruction name (e.g., *mov*), param1 is the first parameter, and param2 is the second parameter. Again, a parameter may be one of {*register*, *memory*, *constant*}. So, the second instruction above: "*or byte [eax+14002700], dl*" becomes "*or.memory.register*" in our representation.

### 3.3 DLL function call feature

Here the granularity level is a DLL function call. An *n*-gram of DLL function call is a sequence of *n* DLL function calls (possibly with other instructions in between two successive calls) in an executable. We extract the information about DLL function calls made by a program from the header of the disassembled file. This is illustrated in Fig. 1. In our experiments, we use only 1-grams of DLL calls, since the higher grams have poorer performance. We enumerate all the DLL function names that have been used by each of the benign and malicious executables, and select the best *S* of them using information gain. We will mention this feature set as *DLL-call feature set* (*DFS*).

## 4 The hybrid feature retrieval model

The HFR Model extracts and combines three different kinds of features, as illustrated in Fig. 2. HFR consists of different phases and components. The feature extraction components have already been discussed in details. Below is a brief description of the model.

### 4.1 Description of the model

The HFR Model consists of two phases: a training phase and a test phase. The training phase is shown in Fig. 2a, and the test phase is shown in Fig. 2b. In the training phase we extract binary *n*-gram features (BFS) and DLL call features (DFS) using the approaches explained in Sections 3.1 and 3.3, respectively. We then apply AFR algorithm (to be explained shortly) to retrieve the derived assembly features (DAFs) that represent the selected binary *n*-gram features. These three kinds of features are combined into the *hybrid feature set*, or HFS in short. Please note that DAF is different from assembly *n*-gram features (i.e., AFS).

AFS are not used in HFS because of our findings that DAF performs better than them. We compute the binary feature vector corresponding to the HFS using the technique explained in Section 4.3, and train a classifier using SVM, boosted decision tree, and other classification methods. In the test phase, we scan each test instance and compute the feature vector corresponding to the HFS. This vector is tested against the classifier. The classifier outputs the class prediction {benign, malicious} of the test file.

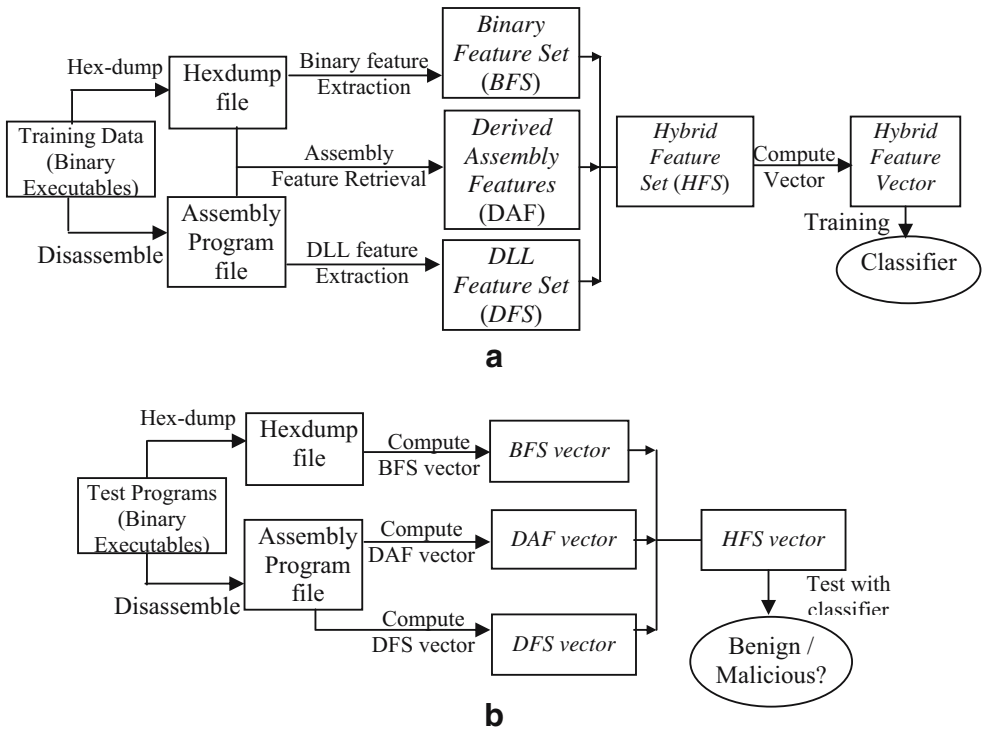


**Fig. 2** The hybrid feature retrieval model, **a** training phase, **b** test phase

### 4.2 The assembly feature retrieval (AFR) algorithm

The AFR algorithm is used to extract assembly instruction sequences (i.e., DAFs) corresponding to the binary *n*-gram features. The main idea is to obtain the complete assembly instruction sequence of a given binary *n*-gram feature. The rationale behind using DAF is as follows. A binary *n*-gram may represent partial information, such as part(s) of one or more assembly instructions or a string inside the code block. We apply AFR algorithm to obtain the complete instruction or instruction sequence (i.e., a DAF) corresponding to the partial one. Thus DAF represents more complete information, which should be more useful in distinguishing the malicious and benign executables. However, binary *n*-grams are still required because they also contain other information like string data, or important bytes at the program header. AFR algorithm consists of several steps. In the first step, a *linear address matching* technique is applied as follows. The offset address of the *n*-gram in the hexdump file is used to find instructions at the same offset at the corresponding assembly program file. Based on the offset value, one of the three situations may occur:

1. The offset is before program entry point, so there is no corresponding assembly code for the *n*-gram. We refer to this address as *address before entry point* (ABEP).
2. There is some data, but no code at that offset. We refer to this address as DATA.
3. There is some code at that offset. We refer to this address as CODE. If this offset is in the middle of an instruction, then we take the whole instruction and consecutive instructions within *n* bytes from the instruction.

In the second step, the best CODE instance is selected among all CODE instances. We apply a heuristic to find the best sequence, called the *most distinguishing instruction sequence* (MDIS) heuristic. According to this heuristic, we choose the instruction sequence that has the highest IG. The AFR algorithm is sketched in Algorithm 3. A comprehensive example of the algorithm is illustrated in Appendix A.

Description of the algorithm: line 1 initializes the lists that would contain the assembly sequences. The *for* loop in line 2 runs for each hexdump file. Each hexdump file is scanned and *n*-grams are extracted (line 4–5). If any of these *n*-grams are in the BFS (line 6–7), then we read the instruction sequence from the corresponding assembly program file at the corresponding address (line 8–10). This sequence is added to the appropriate list (line 12). In this way, we collect all the sequences corresponding to each *n*-gram in the *BFS*. In phase II, we select the best sequence in each *n*-gram list using IG (lines 18–21). Finally, we return the best sequences, i.e., DAFs.

**Algorithm 3** The assembly feature retrieval algorithm

```
Procedure Assembly_Feature_Retrival(G, A, B)
 G = {g1, g2,...,gM}: the selected n-gram features (BFS)
 A = {A1, A2, … , AL} : all Assembly files
 B = {B1, B2, … , BL} : all hexdump files
 S = size of BFS
 L = #of training files
Qi: a list containing the possible instruction sequences for gi
//phase I: sequence collection
1. for i = 1 to S do Qi ← empty  end for //initialize sequence lists
2. for each Bi ∈ B do                  //phase I: sequence collection
3.   offset ← 0                        //current offset in file
4.   while not EOF(Bi) do              //read the whole file
5.       g ← next_ngram(Bi)            //read next n-gram
6.       <index, found> ← BinarySearch(G, g) // seach g in G
7.       if found then  // found
8.          q ← an empty sequence
9.          for each instruction r in Ai with address(r)∈ [offset, offset + n]  do
10.              q ← q ∪ r
11.         end for
12.         Qindex ← Qindex ∪ q //add to the sequence
13.      end if
14.      offset = offset + 1
15.  end while
16. end for
17. V ← empty list                     //phase II: sequence selection
18. for i = 1 to S do                  //for each Qi
19.      q← t ∈ {Qi | ∀u∈ Qi IG(t) >= IG(u) //the sequence with the highest IG
20.      V ← V ∪ q
21  end for
22. return V     // DAF sequences
```

Time complexity of this algorithm is $O(nB\log_2 S)$, where $B$ is the total size of training set in *bytes*, $S$ is the total #of selected binary *n*-gram, and $n$ is size of each *n*-gram in *bytes*. Space complexity is $O(SC)$, where $S$ is defined as above and $C$ is the average #of assembly sequences found per binary *n*-gram. The running time and memory requirements of all three algorithms are summarized in Appendix B.

### 4.3 Feature vector computation and classification

Each feature in a feature set (e.g., HFS, BFS) is a binary feature, meaning, its value is either 1 or 0. If the feature is present in an instance (i.e. an executable), then its value is 1, otherwise its value is 0. For each training (or testing) instance, we compute a feature vector, which is a bit vector consisting of the feature-values of the corresponding feature set. For example, if we want to compute the feature vector $V_{\mathrm{BFS}}$ corresponding to BFS of a particular instance $I$, then for each feature $f \in$ BFS we search $f$ in $I$. If $f$ is found in $I$, then we set $V_{\mathrm{BFS}}[f]$ (i.e., the bit corresponding to $f$) to 1, otherwise, we set it to 0. In this way, we set/reset each bits in the feature vector. These feature vectors are used by the classifiers for training/testing.

We apply SVM, Naïve Bayes (NB), Boosted decision tree, and other classifiers for the classification task. SVM can perform either linear or non-linear classification. The linear classifier proposed by Vladimir Vapnik creates a

hyperplane that separates the data points into two classes with the maximum-margin. A maximum-margin hyperplane is the one that splits the training examples into two subsets, such that the distance between the hyperplane and its closest data point(s) is maximized. A non-linear SVM (Boser et al. 2003) is implemented by applying kernel trick to maximum-margin hyper-planes. The feature space is transformed into a higher dimensional space, where the maximum-margin hyperplane is found. A decision tree contains attribute-tests at each internal node and a decision at each leaf node. It classifies an instance by performing attribute tests from root to a decision node. Decision tree is a rule-based classifier. Meaning, we can obtain human-readable classification rules from the tree. J48 is the implementation of C4.5 Decision Tree algorithm. C4.5 is an extension to the ID3 algorithm invented by Quinlan. A boosting technique called Adaboost combines multiple classifiers by assigning weights to each of them according to their classification performance (Freund and Schapire 1996). The algorithm starts by assigning equal weights to all training samples, and a model is obtained from this training data. Then each misclassified example's weight is increased, and another model is obtained from this new training data. This is iterated for a specified number of times. During classification, each of these models is applied on the test data, and a weighted voting is performed to determine the class of the test instance. We use the AdaBoost.M1 algorithm (Freund and Schapire 1996) on NB, and J48. We only report SVM, and Boosted J48 results because they have the best results. It should be noted that we do not have any preference of any of these two classifiers over the other. We report these accuracies in the results section (Section 5.3).

## 5 Experiments

We design our experiments to run on two different datasets. Each dataset has different sizes and distributions of benign and malicious executables. We generate all kinds of *n*-gram features (e.g. BFS, AFS, DFS) using the techniques explained in Section 3. Notice that the BFS corresponds to the features extracted by Kolter and Maloof's method (Kolter and Maloof 2004). We also generate the DAF and HFS using our model as explained in Section 4. We test the accuracy of each of the feature sets applying a three-fold cross validation using classifiers such as SVM, decision tree, Naïve Bayes, Bayes Net and Boosted decision tree. Among these classifiers, we obtain the best results with SVM and Boosted decision tree, reported in the results section (Section 5.3). We do not report other classifier results due to space limitations. In addition to this, we compute the average accuracy, false positive and false negative rate, and *receiver operating characteristic* (ROC) graphs (using techniques in Fawcett 2003). We also compare the running time and performance/cost trade-off between HFS and BFS.

### 5.1 Dataset

We have two non-disjoint datasets. The first dataset (dataset1) contains a collection of 1,435 executables, 597 of which are benign and 838 are malicious. The second dataset (dataset2) contains 2,452 executables, having 1,370 benign and 1,082 malicious executables. So, the distribution of dataset1 is benign=41.6%, malicious=58.4%, and that of dataset2 is benign=55.9%, malicious=44.1%. This distribution was chosen intentionally to evaluate the performance of the feature sets in different scenarios. We collect the benign executables from different Windows XP, and Windows 2000 machines, and collect the malicious executables from (VX-Heavens 2006), which contains a large collection of malicious executables. The benign executables contain various applications found at the Windows installation folder (e.g. "C:\Windows"), as well as other executables in the default program installation directory (e.g., "C:\Program Files"). Malicious executables contain Viruses, Worms, Trojans, and Back-doors. We select only the Win32 Portable Executables (P.E.) in both the cases. We would like to experiment with the ELF executables in future.

### 5.2 Experimental setup

Our implementation is developed in Java with JDK 1.5. We use the libSVM library (LIBSVM 2006) for running SVM, and Weka ML toolbox (WEKA 2006) for running Boosted decision tree and other classifiers. For SVM, we run C-SVC with a Polynomial kernel; using gamma=0.1, and epsilon=1.0E−12. For Boosted decision tree we run 10 iterations of the AdaBoost algorithm on the C4.5 decision tree algorithm,

**Table 1** Classification accuracy (%) of SVM on different feature sets

| *n* | Dataset1 | | | Dataset2 | | |
|---|---|---|---|---|---|---|
| | HFS | BFS | AFS | HFS | BFS | AFS |
| 1 | 93.4 | 63.0 | 88.4 | 92.1 | 59.4 | 88.6 |
| 2 | 96.8 | 94.1 | 88.1 | 96.3 | 92.1 | 87.9 |
| 4 | 96.3 | 95.6 | 90.9 | 97.4 | 92.8 | 89.4 |
| 6 | 97.4 | 95.5 | 87.2 | 96.9 | 93.0 | 86.7 |
| 8 | 96.9 | 95.1 | 87.7 | 97.2 | 93.4 | 85.1 |
| 10 | 97.0 | 95.7 | 73.7 | 97.3 | 92.8 | 75.8 |
| Avg | 96.30 | 89.83 | 86.00 | 96.20 | 87.25 | 85.58 |
| Avg[a] | 96.88 | 95.20 | 85.52 | 97.02 | 92.82 | 84.98 |

[a] Average accuracy excluding 1-gram

called J48. We set the parameter $S$ (# of selected features) to 500, since it is the best value found in our experiments. Most of our experiments are run on two machines: a Sun Solaris machine with 4 GB main memory and 2 GHz clock speed, and a LINUX machine with 2 GB main memory and 1.8 GHz clock speed. The reported running times are based on the latter machine. The disassembly and hex-dump are done only once for all machine executables and the resulting files are stored. We then run our experiments on the stored files.

### 5.3 Results

In this sub-section, we first report and analyze the results obtained by running SVM on the dataset. Later, we show the accuracies of Boosted J48. Since the results from Boosted J48 are almost the same as SVM, we do not report the analyses based on Boosted J48.

*Accuracy* Table 1 shows the accuracy of SVM on different feature-sets. The columns headed by HFS, BFS, and AFS represent the accuracies of the Hybrid Feature Set (our method), Binary Feature Set (Kolter and Maloof's feature set) and Assembly Feature Set, respectively. Note that the AFS is different from the DAF (i.e., derived assembly features) that has been used in the HFS (see Section 4.1 for details). Table 1 reports that the classification accuracy of HFS is always better than other models, on both datasets. It is interesting to note that the accuracies for 1-gram BFS are very low in both datasets. This is because 1-gram is only a 1-byte long pattern, having only 256 different possibilities. Thus, this pattern is not useful at all in distinguishing the malicious executables from the normal, and may not be used in a practical application. So, we exclude the 1-gram accuracies while computing the average accuracies (i.e., the last row).

*Dataset1* Here the best accuracy of the hybrid model is for $n$=6, which is 97.4, and is the highest among all feature sets. On average, the accuracy of HFS is 1.68% higher than that of BFS, and 11.36% higher than that of AFS. Accuracies of AFS are always the lowest. One possible reason behind this poor performance is that AFS considers only the CODE (see Section 4.2) part of the executables. So, AFS misses any distinguishing pattern carried by the ABEP or DATA parts, and as a result, the extracted features have poorer performance. Moreover, the accuracy of AFS greatly deteriorates for $n$=10. This is because longer sequences of instructions are rarer in either class of executables (malicious/benign), so these sequences have less distinguishing power. On the other hand, BFS considers all parts of the executable, achieving higher accuracy. Finally, HFS considers DLL calls, as well as BFS and DAF. So, HFS has better performance than BFS.

*Dataset2* Here the differences between the accuracies of HFS and BFS are greater than that of dataset1. The average accuracy of HFS is 4.2% higher than that of BFS. Accuracies of AFS are again the lowest. It is interesting to note that HFS has an improved performance over BFS (and AFS) in dataset2. Two important conclusions may be derived from this observation. First, dataset2 is much larger than dataset1, having more diverse set of examples. Here HFS performs better than dataset1, whereas BFS performs worse than dataset1. This implies that HFS is more robust than BFS in a diverse and larger set of instances. Thus, HFS is more applicable than BFS in a large, diverse corpus of executables. Second, dataset2 has more benign executables than malicious, whereas dataset1 has less benign executables. This distribution of dataset2 is more likely in a real world, where benign executables outnumber malicious executables. This implies that HFS is likely to perform better than BFS in a real-world scenario, having larger number of benign executables in the dataset.

*Statistical significance test* We also perform a pair-wise two-tailed $t$-test on the HFS and BFS accuracies to test whether the differences between their accuracies are statistically significant. We exclude 1-gram accuracies from this test for the reason explained above. The result of the $t$-test is summarized in Table 2. The $t$-value shown in this table is the value of $t$ obtained from the accuracies. There are (5+5−2) *degrees of freedom*, since we have five observations in each group, and there are two groups (i.e., HFS and BFS). *Probability* denotes the probability of rejecting the NULL hypothesis (that there is no difference between HFS and BFS accuracies), while $p$-value denotes the probability of accepting the NULL hypothesis. For dataset1, the probability is 99.65%, and for dataset2, it is 100.0%. Thus, we conclude that the average accuracy of HFS is significantly higher than that of BFS.

*DLL call feature* Here we report the accuracies of the DLL function call features (DFS). The 1-gram accuracies are: 92.8% for dataset1 and 91.9% for dataset2. The accuracies for higher grams are less than 75%, so we do not report them. The reason behind this poor performance is possibly that there are no distinguishing call-sequences that can identify the executables as malicious or benign.

**Table 2** Pair-wise two-tailed $t$-test results comparing HFS and BFS accuracies

| | DataSet1 | DataSet2 |
|---|---|---|
| $t$-value | 8.9 | 14.6 |
| Degrees of freedom | 8 | 8 |
| Probability | 0.9965 | 1.00 |
| $p$-value | 0.0035 | 0.0000 |

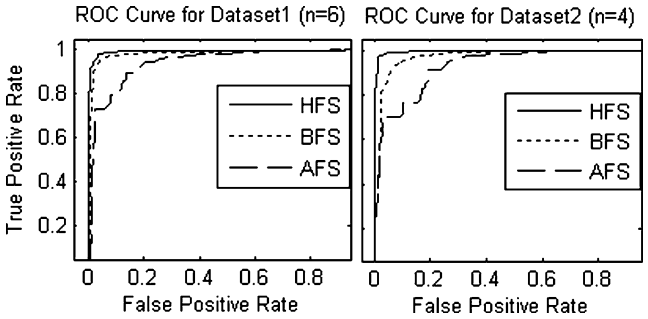


**Fig. 3** ROC curves for different feature sets in dataset1 (*left*), and dataset2 (*right*)

*ROC curves* ROC curves plot the true positive rate against the false positive rates of a classifier. Figure 3 shows ROC curves of dataset1 for $n$=6 and dataset2 for $n$=4 based on SVM testing. ROC curves for other values of $n$ have similar trends, except for n=1, where AFS performs better than BFS. It is evident from the curves that HFS is always dominant (i.e. has larger area under the curve) over the other two and it is more dominant in dataset2. Table 3 reports the area under the curve (AUC) for the ROC curves of each of the features sets. A higher value of AUC indicates a higher probability that a classifier will predict correctly. Table 3 shows that the AUC for HFS is the highest, and it improves (relative to other two) in dataset2. This also supports our hypothesis that our model will perform better in a more likely real-world scenario, where benign executables occur more frequently.

*False positive and false negative* Table 4 reports the false positive and false negative rates (in percentage) for each feature set based on SVM output. The last row reports the average. Again, we exclude the 1-gram values from the average. Here we see that in dataset1, the average false positive rate of HFS is 4.9%, which is the lowest. In dataset2, this rate is even lower (3.2%). False positive rate is a measure of false alarm rate. Thus, our model has the lowest false alarm rate. We also observe that this rate decreases as we increase the number of benign examples. This is because the classifier gets more familiar with benign executables and misclassifies fewer of them as malicious. We believe that a large collection of training set with a larger portion of benign executables would eventually diminish false positive rate towards zero. The false negative rate is also the lowest for HFS as reported in Table 4.

**Table 3** Area under the ROC curve on different feature sets

| $n$ | Dataset1 | | | Dataset2 | | |
|---|---|---|---|---|---|---|
| | HFS | BFS | AFS | HFS | BFS | AFS |
| 1 | 0.9767 | 0.7023 | 0.9467 | 0.9666 | 0.7250 | 0.9489 |
| 2 | 0.9883 | 0.9782 | 0.9403 | 0.9919 | 0.9720 | 0.9373 |
| 4 | 0.9928 | 0.9825 | 0.9651 | 0.9948 | 0.9708 | 0.9515 |
| 6 | 0.9949 | 0.9831 | 0.9421 | 0.9951 | 0.9733 | 0.9358 |
| 8 | 0.9946 | 0.9766 | 0.9398 | 0.9956 | 0.9760 | 0.9254 |
| 10 | 0.9929 | 0.9777 | 0.8663 | 0.9967 | 0.9700 | 0.8736 |
| Avg | 0.9900 | 0.9334 | 0.9334 | 0.9901 | 0.9312 | 0.9288 |
| Avg[a] | 0.9927 | 0.9796 | 0.9307 | 0.9948 | 0.9724 | 0.9247 |

[a] Average value excluding 1-gram

**Table 4** False positive and false negative rates on different feature sets

| $n$ | Dataset1 | | | Dataset2 | | |
|---|---|---|---|---|---|---|
| | HFS | BFS | AFS | HFS | BFS | AFS |
| 1 | 8.0/5.6 | 77.7/7.9 | 12.4/11.1 | 7.5/8.3 | 65.0/9.8 | 12.8/9.6 |
| 2 | 5.3/1.7 | 6.0/5.7 | 22.8/4.2 | 3.4/4.1 | 5.6/10.6 | 15.1/8.3 |
| 4 | 4.9/2.9 | 6.4/3.0 | 16.4/3.8 | 2.5/2.2 | 7.4/6.9 | 12.6/8.1 |
| 6 | 3.5/2.0 | 5.7/3.7 | 24.5/4.5 | 3.2/2.9 | 6.1/8.1 | 17.8/7.6 |
| 8 | 4.9/1.9 | 6.0/4.1 | 26.3/2.3 | 3.1/2.3 | 6.0/7.5 | 19.9/8.6 |
| 10 | 5.5/1.2 | 5.2/3.6 | 43.9/1.7 | 3.4/1.9 | 6.3/8.4 | 30.4/16.4 |
| Avg | 5.4/2.6 | 17.8/4.7 | 24.4/3.3 | 3.9/3.6 | 16.1/8.9 | 18.1/9.8 |
| Avg[a] | 4.9/2.0 | 5.8/4.1 | 26.8/1.7 | 3.2/2.7 | 6.3/8.1 | 19.2/17.8 |

[a] Average value excluding 1-gram

*Running Time* We compare in Table 5 the running times (feature extraction, training, testing) of different kind of features (HFS, BFS, AFS) for different values of $n$. Feature extraction time for HFS and AFS includes the disassembly time, which is 465 s (in total) for dataset1, and 865 s (in total) for dataset2. Training time is the sum of feature extraction time, feature-vector computation time, and SVM training time. Testing time is the sum of disassembly time (except BFS) feature-vector computation time, and SVM classification time. Training and testing times based on Boosted J48 have almost similar characteristics, so we do not report them. Table 5 also reports the cost factor as a ratio of time required for HFS relative to BFS. The column *cost factor* shows this comparison. The average feature-extraction times are computed by excluding the 1- and 2-gram, since these grams are unlikely to be used in practical applications. The boldface cells in the table are of particular interest to us. From the table we see that the running times for HFS training and testing on dataset1 are 1.17 and 4.87 times higher than those of BFS, respectively. For dataset2, these numbers are 1.08 and 4.5, respectively. The average throughput for HFS is found to be 0.6 MB/sec (in both datasets), which may be considered as near real-time performance. Finally, we summarize the cost/performance trade-off in Table 6. The column *Performance improvement* reports the accuracy improvement of HFS over BFS. The cost factors are shown in the next two columns. If we drop the disassembly time from testing time (considering that

**Table 5** Running times (in seconds)

| | *n* | Dataset1 | | | | Dataset2 | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | HFS | BFS | AFS | Cost factor[a] | HFS | BFS | AFS | Cost factor[a] |
| Feature extraction | 1 | 498.41 | 135.94 | 553.2 | 3.67 | 841.67 | 166.87 | 908.42 | 5.04 |
| | 2 | 751.93 | 367.46 | 610.85 | 2.05 | 1,157.5 | 443.99 | 949.7 | 2.61 |
| | 4 | 1,582.21 | 1,189.65 | 739.51 | 1.33 | 3,820.7 | 3,103.14 | 1,194.4 | 1.23 |
| | 6 | 2,267.94 | 1,877.6 | 894.26 | 1.21 | 8,010.24 | 7,291.4 | 1,519.56 | 1.1 |
| | 8 | 2,971.9 | 2,572.26 | 1,035.06 | 1.16 | 11,736. | 11,011.67 | 1,189.01 | 1.07 |
| | 10 | 3,618.31 | 3,223.21 | 807.85 | 1.12 | 15,594. | 14,858.68 | 2,957 | 1.05 |
| | Avg[b] | 2,610.09 | 2,215.68 | 869.17 | 1.18 | 9,790.6 | 9,066.22 | 1,714.99 | 1.08 |
| Training | Avg[c] | 2,654.68 | 2,258.86 | 910.68 | **1.18** | 9,857.85 | 9,134.36 | 1,782.8 | **1.08** |
| Testing | Avg[c] | 195.25 | 40.09 | 194.9 | **4.87** | 377.89 | 83.91 | 348.35 | **4.5** |
| Testing/MB MB | | **1.74** | **0.36** | 1.74 | 4.87 | **1.57** | **0.35** | 1.45 | 4.5 |
| Throughput(MB/s)) | | **0.6** | **2.8** | 0.6 | – | **0.64** | **2.86** | 0.69 | – |

[a] Ratio of time required for HFS to time required for BFS
[b] Average feature extraction times excluding 1- and 2-gram
[c] Average training/testing times excluding 1- and 2-gram

disassembly is done offline), then the testing cost factor diminishes to 1.0 for both dataset. It is evident from Table 6 that the performance/cost trade-off is better for dataset2 than dataset1. Again, we may infer that our model is likely to perform better in a larger and more realistic dataset. The main bottleneck of our system is disassembly cost. The testing cost factor is higher because here larger proportion of time is used up in disassembly. We believe that this factor may be greatly reduced by optimizing the disassembler, and considering that disassembly can be done offline.

*Training & Testing with Boosted J48* We also train and test with this classifier and report the classification accuracies for different features and different values of *n* in Table 7. The second last row (*Avg*) of Table 7 is the average of 2- to 10-g accuracies. Again, for consistency, we exclude 1-gram from the average. We also include the average accuracies of SVM (from last row of Table 1) in the last row of Table 7 for ease of comparison. We would like to point out some important observations regarding this comparison. First, the average accuracies of SVM and Boosted J48 are almost the same, being within 0.4% of each other (for HFS). There is no clear winner between these two classifiers. So, we may use any of these classifiers for our model. Second, accuracies of HFS are again the best among all three. Besides, HFS has 1.84% and 3.6% better accuracies than BFS in dataset1 and dataset2, respectively. This result also justifies our claim that HFS is a better feature set than BFS, irrespective of the classifier used.

**Table 6** Performance/cost trade-off between HFS and BFS

| | Performance improvement (%) (HFS–BFS) /BFS | Training cost factor (HFS/BFS) | Testing cost factor (HFS/BFS) |
|---|---|---|---|
| Dataset1 | 1.73 | 1.17 | 4.87 |
| Dataset2 | 4.52 | 1.08 | 4.5 |

## 6 Conclusion

Our HFR model is a novel idea in malicious code detection. It extracts useful features from disassembled executables using the information obtained from binary executables. It then combines the assembly features with other features like DLL function calls and binary *n*-gram features. We have addressed a number of difficult implementation issues and provided efficient, scalable and practical solutions. The difficulties that we face during implementation are related to memory limitations and long running times. By using

**Table 7** Classification accuracy (%) of boosted J48 on different feature sets

| *N* | Dataset1 | | | Dataset2 | | |
|---|---|---|---|---|---|---|
| | HFS | BFS | AFS | HFS | BFS | AFS |
| 1 | 93.9 | 64.1 | 91.3 | 93.5 | 58.8 | 90.2 |
| 2 | 96.4 | 93.2 | 89.4 | 97.1 | 92.7 | 85.1 |
| 4 | 96.3 | 95.4 | 92.1 | 97.2 | 93.6 | 87.5 |
| 6 | 96.3 | 95.3 | 87.8 | 97.6 | 93.6 | 85.4 |
| 8 | 96.7 | 94.1 | 89.1 | 97.6 | 94.3 | 83.7 |
| 10 | 96.6 | 95.1 | 77.1 | 97.8 | 95.1 | 82.6 |
| Avg[a] (Boosted J48) | 96.46 | 94.62 | 87.1 | 97.46 | 93.86 | 84.86 |
| Avg[b](SVM) | 96.88 | 95.20 | 85.52 | 97.02 | 92.82 | 84.98 |

[a] Average accuracy excluding 1-gram
[b] Average accuracy for SVM (from Table 1)

efficient data structures, algorithms and disk I/O, we are able to implement a fast, scalable and robust system for malicious code detection. We run our experiments on two datasets with different class distribution, and show that a more realistic distribution improves the performance of our model.

Our model also has a few limitations. First, it does not directly handle obfuscated DLL calls or encrypted/packed binaries. There are techniques available for detecting obfuscated DLL calls in the binary (Lakhotia et al. 2005), and to unpack the packed binaries automatically (Royal et al. 2006). We may apply these tools for de-obfuscation/decryption and use their output to our model. Although this is not implemented yet, we look forward to integrate these tools with our model in our future versions. Second, the current implementation is an offline detection mechanism. Meaning, it cannot be directly deployed on a network to detect malicious code. However, it can detect malicious codes in near real time.

We address these issues in our future work, and vow to solve these problems. We also propose several modifications to our model. For example, we would like to combine our features with run-time characteristics of the executables. Besides, we propose building a feature-database that would store all the features and be updated incrementally. This would save a large amount of training time and memory.

**Table 8** Assembly code sequence for binary 4-g "00005068"

| Sequence # | Op-code | Assembly code | Information gain |
|---|---|---|---|
| 1 | E8B702**0000** | call 00401818 | 0.5 |
| | **50** | push eax | |
| | **68**28234000 | push 00402328 | |
| 2 | 0FB6800D02**0000** | movzx eax,byte [eax+20] | 0.1 |
| | **50** | push eax | |
| | **68**CC000000 | push 000000CC | |
| 3 | 8B805C04**0000** | mov eax, dword [eax+45] | 0.2 |
| | **50** | push eax | |
| | **68**01040000 | push 00000401 | |
| 29 | 8D8010010**000** | lea eax, dword [eax+110] | 0.7 |
| | **50** | push eax | |
| | **68**07504000 | push 00405007 | |
| 50 | 25FFFF**0000** | and eax, 0000FFFF | 0.3 |
| | **50** | push eax | |
| | **68**E8164100 | push 004116E8 | |
| 90 | 25FFFF**0000** | and eax, 0000FFFF | 0.4 |
| | **50** | push eax | |
| | **68**600E4100 | push 00410E60 | |

**Table 9** Time and space complexities of different algorithms

| Algorithm | Time complexity | Space complexity |
|---|---|---|
| Feature collection | $O(B\log_2 K)+O(N)$ | $O(K)$ |
| Feature selection | $O(N\log_2 S)$ | $O(S)$ |
| Assembly feature retrieval | $O(nB\log_2 S)$ | $O(SC)$ |
| Total (worst case) | $O(nB\log_2 K)$ | $O(SC)$ |

**Acknowledgment** The work reported in this paper is supported by AFOSR under contract FA9550-06-1-0045 and by the Texas Enterprise Funds. We thank Dr. Robert Herklotz of AFOSR and Prof. Robert Helms, Dean of the School of Engineering at the University of Texas at Dallas for funding this research.

## Appendix A

Here we illustrate an example run of the AFR algorithm. The algorithm scans through each hexdump file, sliding a window of $n$ bytes and checking the $n$-gram against the binary feature set (BFS). If a match is found, then we collect the corresponding (same offset address) assembly instruction sequence in the assembly program file. In this way, we collect all possible instruction sequences of all the features in BFS. Later, we select the best sequence using information gain. *Example-III*: Table 8 shows an example of the collection of assembly sequences and their IG values corresponding to the $n$-gram "00005068." Note that this $n$-gram has 90 occurrences (in all hexdump files). We have shown only 5 of them for brevity. The bolded portion of the op-code in Table 8 represents the $n$-gram. According to the *Most Distinguishing Instruction Sequence* (MDIS) heuristic, we find that sequence #29 attains the highest information gain, which is selected as the DAF of the $n$-gram. In this way, we select one DAF per binary $n$-gram, and return all DAFs.

## Appendix B

Here we summarize the time and space complexities of our algorithms in Table 9.

$B$ is the total size of training set in *bytes*, $C$ is the average #of assembly sequences found per binary $n$-gram, $K$ is the maximum #of nodes of the AVL tree (i.e., threshold), $N$ is the total number of $n$-grams collected, $n$ is size of each $n$-gram in *bytes,* and $S$ is the total number of selected $n$-grams. The worst case assumption: $B > N$ and $SC > K$

## References

Boser, B. E., Guyon, I. M., & Vapnik, V. N. (2003). A training algorithm for optimal margin classifiers. In D. Haussler (Ed.), *5th annual ACM workshop on COLT* (pp. 144–152). New York: ACM Press.

Cygnus (1999). *GNU Binutils Cygwin*. Retrieved from http://sourceware.cygnus.com/cygwin.

Fawcett, T. (2003). *ROC Graphs: Notes and practical considerations for researchers*. Tech Report HPL-2003-4, HP Laboratories. Retrieved May 26, 2006, from http://www.hpl.hp.com/personal/TomFawcett/papers/ROC101.pdf.

Freund, Y., & Schapire, R. (1996). Experiments with a new boosting algorithm. In *Proc. of the thirteenth international conference on machine learning* (pp. 148–156). San Mateo, CA: Morgan Kaufmann.

Garg, A., Rahalkar, R., Upadhyaya, S., & Kwiat, K. (2006). Profiling users in GUI based systems for masquerade detection. In *Proc. of the 7th IEEE information assurance workshop (IAWorkshop 2006)* (pp. 48–54).

Golbeck, J., & Hendler, J. (2004). *Reputation network analysis for email filtering*. In CEAS.

GoodRich, M. T., & Tamassia, R. (2006). *Data structures and algorithms in Java* (4th ed.). New York: Wiley.

LIBSVM. (2006). *A library for support vector machine.* Retrieved June 1, 2006 from http://www.csie.ntu.edu.tw/~cjlin/libsvm/.

Kim, H. A., & Karp, B. (2004). Autograph: Toward automated, distributed worm signature detection. In *Proc. of the 13th Usenix security symposium (Security 2004)* (pp. 271–286).

Kolter, J. Z., & Maloof, M. A. (2004). Learning to detect malicious executables in the wild. In *Proc. of the tenth ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 470–478).

Lakhotia, A., Kumar, E. U., & Venable, M. (2005). A method for detecting obfuscated calls in malicious binaries. *IEEE Transactions on Software Engineering, 31*(11), 955–968.

Masud, M. M., Khan, L., & Thuraisingham, B. (2007a). Feature based techniques for auto-detection of novel email worms. In *Proc. of the eleventh Pacific-Asia conference on knowledge discovery and data mining (PAKDD'07)* (pp. 205–216). LNAI 4426/2007.

Masud, M. M., Khan, L., & Thuraisingham, B. (2007b). A hybrid model to detect malicious executables. In *Proc. of the IEEE international conference on communication (ICC'07)* (pp. 1443–1448).

Mitchell, T. (1997). *Machine learning*. New York: McGraw-Hill.

Newman, M. E. J., Forrest, S., & Balthrop, J. (2002). Email networks and the spread of computer viruses. *Physical Review, 66*(3), 035101.

Newsome, J., Karp, B., & Song, D. (2005). Polygraph: Automatically generating signatures for polymorphic worms. In *Proc. of the IEEE symposium on security and privacy* (pp. 226–241).

Royal, P., Halpin, M., Dagon, D., Edmonds, R., & Lee, W. (2006). PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *Proc. of 22nd annual computer security applications conference (ACSAC'06)* (pp. 289–300).

Schultz, M., Eskin, E., & Zadok, E. (2001a). MEF Malicious email filter, a UNIX mail filter that detects malicious windows executables. In *Proc. of the USENIX annual technical conference—FREENIX track* (pp. 245–252).

Schultz, M., Eskin, E., Zadok, E., & Stolfo, S. (2001b). Data mining methods for detection of new malicious executables. In *Proc. of the IEEE symposium on security and privacy* (pp. 178–184).

Singh, S., Estan, C., Varghese, G., & Savage, S. (2003). *The earlyBird system for real-time detection of unknown worms*. Technical report—cs2003-0761, UCSD.

VX-Heavens. (2006). Retrieved May 6, 2006 from http://vx.netlux.org/.

WEKA. (2006). Retrieved Aug 1, 2006 from http://www.cs.waikato.ac.nz/ml/weka/.

Windows P.E. Disassembler. (1998). Retrieved June 05, 2006 from http://www.geocities.com/~sangcho/index.html.

**Mr. Mohammad Mehedy Masud** is a Ph.D. student at the department of Computer Science at the University of Texas at Dallas (UTD) since August, 2005. He received his undergraduate degree in Computer Science and Engineering (CSE) from Bangladesh University of Engineering and Technology (BUET) in 2001. Before joining UTD, he was a lecturer at the department of CSE, BUET, from 2001 to 2004. He is currently an Assistant Professor (on leave) at the same department. He is a member of the IEEE Computer Society, and the Association for Computing Machinery (ACM). His current areas of research are data mining, intrusion detection, and network security. He has already published 15 conference papers and journal articles and more papers are currently under review. He is also an award-winning programmer at the ACM-International Collegiate Programming Contests (ICPC) World Finals-1999, held in Eindhoven, The Netherlands.

**Dr. Latifur R. Khan** is currently an Associate Professor in the Computer Science department at the University of Texas at Dallas (UTD), where he has taught and conducted research since September 2000. He received his Ph.D. and M.S. degrees in Computer Science from the University of Southern California, in August of 2000, and December of 1996 respectively. He obtained his B.Sc. degree in Computer Science and Engineering from Bangladesh University of Engineering and Technology, Dhaka, Bangladesh in November of 1993. Dr. Khan is the director of the UTD Data Mining/Database Laboratory. Dr. Khan's research areas cover data mining, multimedia information management, and semantic web and database systems. He has served as a committee member in numerous prestigious conferences, symposiums and workshops including the ACM SIGKDD Conference on Knowledge Discovery and Data Mining. Dr. Khan currently serves on the editorial board of North Holland's Computer Standards and Interface Journal, Elsevier Publishing. Dr. Khan has published over 80 papers in prestigious journals and conferences.

**Dr. Bhavani Thuraisingham** is a Professor of Computer Science and the Director of Cyber Security Research Center at the University of Texas at Dallas (UTD). Prior to joining UTD, she was a program director for 3 years at the National Science Foundation (NSF) in Arlington, VA. She has also worked for the Computer Industry in Mpls, MN for over 5 years and has served as an adjunct professor of computer science and member of the graduate faculty at the University of Minnesota and later taught at Boston University. Dr. Thuraisingham's research interests are in the area of Information Security and data management. She has published over 300 research papers including over 60 journals articles and is the inventor of three patents. She serves on the editorial board of numerous journals including ACM Transactions on Information and Systems Security and IEEE Transactions on Dependable and Secure Computing.