J Comput Virol Hack Tech (2013) 9:77–93
DOI 10.1007/s11416-013-0181-8

ORIGINAL PAPER

HDM-Analyser: a hybrid analysis approach based on data mining techniques for malware detection

Mojtaba Eskandari · Zeinab Khorshidpour · Sattar Hashemi

Received: 22 May 2012 / Accepted: 29 January 2013 / Published online: 17 February 2013
© Springer-Verlag France 2013

Abstract Today's security threats like malware are more sophisticated and targeted than ever, and they are growing at an unprecedented rate. To deal with them, various approaches are introduced. One of them is Signature-based detection, which is an effective method and widely used to detect malware; however, there is a substantial problem in detecting new instances. In other words, it is solely useful for the second malware attack. Due to the rapid proliferation of malware and the desperate need for human effort to extract some kinds of signature, this approach is a tedious solution; thus, an intelligent malware detection system is required to deal with new malware threats. Most of intelligent detection systems utilise some data mining techniques in order to distinguish malware from sane programs. One of the pivotal phases of these systems is extracting features from malware samples and benign ones in order to make at least a learning model. This phase is called "Malware Analysis" which plays a significant role in these systems. Since API call sequence is an effective feature for realising unknown malware, this paper is focused on extracting this feature from executable files. There are two major kinds of approach to analyse an executable file. The first type of analysis is "Static Analysis" which analyses a program in source code level. The second one is "Dynamic Analysis" that extracts features by observing program's activities such as system requests during its execution time. Static analysis has to traverse the program's execution path in order to find called APIs. Because it does not have sufficient information about decision making points in the given executable file, it is not able to extract the real sequence of called APIs. Although dynamic analysis does not have this drawback, it suffers from execution overhead. Thus, the feature extraction phase takes noticeable time. In this paper, a novel hybrid approach, HDM-Analyser, is presented which takes advantages of dynamic and static analysis methods for rising speed while preserving the accuracy in a reasonable level. HDM-Analyser is able to predict the majority of decision making points by utilising the statistical information which is gathered by dynamic analysis; therefore, there is no execution overhead. The main contribution of this paper is taking accuracy advantage of the dynamic analysis and incorporating it into static analysis in order to augment the accuracy of static analysis. In fact, the execution overhead has been tolerated in learning phase; thus, it does not impose on feature extraction phase which is performed in scanning operation. The experimental results demonstrate that HDM-Analyser attains better overall accuracy and time complexity than static and dynamic analysis methods.

M. Eskandari (✉) · Z. Khorshidpour · S. Hashemi
Department of Computer Science and Engineering,
Shiraz University, Shiraz, Iran
e-mail: m_eskandari@cse.shirazu.ac.ir

Z. Khorshidpour
e-mail: zkhorshidpur@cse.shirazu.ac.ir

S. Hashemi
e-mail: s_hashemi@cse.shirazu.ac.ir

1 Introduction

Malware, stands for malicious software, is a computer program which has destructive purposes such as viruses, Trojan horses, Spyware, Internet worms etc. Growing the Internet significantly increases the variety and complexity of malware; hence, malware poses serious security effects on information societies. More information about malware can be achieved through [13,25].

In the earlier times, the most effective and accessible method for capturing malware was signature-based detection. These days, this method is known as an industrial standard for detecting malware which is widely utilised by

anti-virus software [1]. Detecting a malware by this method is consisted of several phases. The first phase is extracting at least a signature from a malicious file. A signature can be simply a checksum or a sequence of bytes which is extracted from content of a malicious file. The next phase is publishing signature for usage of end user's anti-virus software. Anti-virus updates its signature database with new signatures. After finishing the update stage, anti-virus is able to detect new malware. This approach is quite fast and effective for detecting known malware.

There are several signature types and extraction methods for various type of malware in terms of infection and self-protection mechanisms. For detection of those malware which do not utilise any obfuscation method in their infection procedure, a checksum of entire file is sufficient because their new generated sample and their ancestors are exactly same. Unfortunately, these days, this kind of malware is rarely found unless it is quite weak and will be stopped in earlier minutes of its propagation. In other words, different tricks have been invented and employed by malware writers to bypass various detection methods. One of the most popular tricks which was utilised in earlier times is altering some sections of an executable file in order to bypass the signature detection scanners. This method has been being used by many malware writers. Luckily, the majority of these malware instances have at least one unchangeable section and this matter can be easily exploited to extract a signature from such malicious file. At this time, one further signature type is introduced which is called "section based checksum". This type of signature is a calculated checksum of at least one unalterable section of an malicious executable file. It is interesting to notice that by applying this method on anti-virus scanner engines, a vast amount of known executable malware including most of Worms, Trojans, Rootkits etc can be detected. The latest practical signature which is used in anti-virus scanners is called "classic signature" throughout this paper. A classic signature is consisted of one or more than one byte sequences which is extracted by a human expert from a infected or a portable executable malicious file. This kind of signature is appropriate for those malware which modify all their sections. To deal with these kinds of malware, experts have to find some sub-sequences which are remain immutable during infection procedure and use them as signature to detect all variants of considered malware. These three type of signature are the most common kinds of signature that use by most of anti-virus scanners. There are some other varieties of signatures that produced by blending these types such as "regular expression" signature which is a classic signature that has some variables within it in order to cover changeable parts of malware.

Despite of effectiveness and reliableness of signature based method, it has two major drawbacks. Signature extraction is a hand crafted and semi-automated task which brings significant difficulties to the scanner engines based on signature database. As a matter of fact, extracting some types of signature especially classic and regular expressions is a tedious task and needs remarkable amount of time and expertise. On the other hand, malware writers deploy diverse self protection techniques and use them in their malware; therefore, it becomes increasingly hard for experts to extract signature for them. The second issue is this method detects malware based on a signature database. In other words, it is able to catch malware which are identified before; thus, it is not a reliable approach for preventing malware attacks in the earliest occurrences.

Two interesting kinds of malware which attempt to be hidden from detectors' sight are Polymorphic and Metamorphic malware. The body of them is modified during infection process. A polymorphic malware has an embedded encryption engine and uses that with a random key to generate a new instance of itself. The content of the new instance is quite different from its original version. This content is decrypted in execution time by a decryption engine. Obviously, it is so hard to detect this type of malware by classic signature base method. In some samples decryption engine has a plain code and this is a pleasant fortune to extract signature from this section; however, in most sophisticated polymorphic malware this part uses some obfuscation techniques in order to avoid detection scanners. The second one is Metamorphic malware, which is able to generate an instance of itself with a different internal structure and probably disparate code; however, it behaves like its ancestors. It employs various obfuscation techniques to alter its code. These two types of malware are big obstacles on the way of signature detection method.

To resolve the main drawback of classic signature based method, which is an inability in detecting unknown malware, behaviour based (or heuristic) solutions are developed. These methods attempt to find the pattern of malware behaviour for further recognitions of similar malicious behaviour. They are able to fulfil the malicious activity detection during code execution by trying to trace any suspicious stir [29]. Even if a code is obfuscated, its behaviour and functionalities remain invariant.

A heuristic approach typically utilises data mining as well as machine learning techniques; hence, it requires feature extraction phase to extract information from executable files. The procedure of feature extraction phase is usually called "malware analysis". There are three major approaches to do that [13]. First one is "Dynamic Analysis" that extracts features by observing program's activities such as system requests during its execution time. The most appropriate feature which is extracted by this analysis method is a sequence of Application Programming Interface (API) calls. API is an interface provided by the operating system for programs

through which they invoke the operating system to get a service [16]. There are other kinds of feature which are used by intelligent malware detection approaches such as DLL usage information, n-grams, assembly instructions sequence, dependency graph, and control flow graph. From all of them, a sequence of called APIs is one of the most effective feature being able to describe the behaviour of a program almost perfectly. The second type of analysis is "Static Analysis", or reverse engineering, which analyses a program in source code level. Assembly code or binary code of a program is investigated in order to find some useful features for distinguishing malware samples from benign ones. Finally, the last type is hybrid approach combining both static and dynamic analyses [21,26].

Static analysis offers information about control flow, data flow and other statistical features of programs without actually running them. Several reverse engineering techniques, including disassembling, decompiling etc., can be applied on binary codes in order to construct an intermediate representation model for them. The most remarkable advantage of static analysis is being free from the execution overhead; nevertheless, it gives merely an approximation of the actual execution because at any decision point, it is solely able to guess which branch will be followed at the execution time [23]. Dynamic analysis runs the given executable file and monitors it's behaviour in a real or a virtual environment. Although it gives the actual information about the control and data flow, this kind of analysis needs a noticeable amount of time for doing the analysis operation.

Since static analysis methods do not run the given programs, they are faster and safer than dynamic ones. However, it is responsible for inability to realise the correct way in decision making points when traversing a control flow graph. Although dynamic analysis does not have this problem, a crucial question is arisen here is in which environment the malicious executable files should be executed. Using a dedicated stand alone machine and restoring the operating system to the first status after each dynamic analysis run might be the first solution; however, it is not an efficient solution because of heavy overhead. A popular alternative is running executable files in a virtual machine; therefore, only the virtual computer would be affected [2]. In the recent times, hybrid analysis has been presented where a combination of static and dynamic information are used to analyse malware samples. This approach tends to take advantages of static and dynamic analysis methods which are speed and accuracy respectively.

In this paper, a novel analysis approach is introduced for extracting the API call sequence from executable files. This approach, HDM-Analyser, analyses a *PE-file*, which is the standard executable (EXE) file format used by the Microsoft Windows operating system, statically and simultaneously utilises runtime information on ambiguous points to make them clear. HDM-Analyser is faster and safer than dynamic analysis method and has better accuracy than static analysis approach; thus, it can be used as feature extractor module in practical malware detection systems.

Since it is essential for end users to work in a safe environment while the system's performance is not declined by the employed malware detection system, preparing a precise and quick malware analysis method is an inevitable matter, which is almost a heavy procedure in the detecting process in terms of time complexity. Consequently, these two measures is chosen as criteria to compare performance of the presented approach with its rivals.

The paper is organised as follows: in Sect. 2, a variety of data-mining based malware detection approaches are presented. As mentioned, most of these approaches need an analysis procedure to extract the required features. HDM-Analyser system is expressed in Sect. 3. The experimental results are described and discussed in Sect. 4. The paper is concluded in the final section.

2 Related work

Since this paper is focused on malware analysis particularly extracting API calls, in this section malware detection approaches in which malware analysis procedure is used are described. Malware analysis has been under investigation since the early days of vicious activity appearance. [13] is one of the original works that presents a great overview on vast majority of malware kinds, their origination, their purpose, and their associated detection methods.

The analysis can be performed at the source code level or at the binary level where merely the executable file is accessible. It is unrealistic to assume the availability of source code for every program [4]; however, there are some tools being able to generate an approximation source code out of binary executable files such as decompilers and disassemblers. The required features from the executables can be extracted either by executing them or performing static reverse engineering or by using both techniques, which is known as hybrid analysis.

Dynamic analysis requires to run the program and monitor its execution in a real or a virtual machine environment. One of the earliest system call (API) detection engines has been created by Sekar et al. [22]. It is able to compare APIs which are modelled previously with the APIs called at runtime. An intermediate representation of an executable file is constructed in the form of an Auditing Specification Language (ASL). The ASL is compiled into a *C++ class* that is then compiled and linked to create a model of an API in the system call detection engine. In fact, it is an anomaly-based detector wherein the APIs required by a program are manually specified and the detector monitors API calls, flagging any calls outside the specification as malicious. This method

has to compile and link the ASL for each sample; thus, it seems this approach needs considerable amount of time for its detection process. Unfortunately, there are no reasonable experimental results on their paper.

Roundy and Miller have introduced a pre-execution analysis by combining static and dynamic methods to conduct control and data-flow analysis [21]. These analyses form the interface by which the analyst instruments the code. This interface simplifies the instrumentation task and reduce the number of instrumented program locations by a hundred-fold relative to existing instrumentation-based methods of identifying unpacked code. They showed that by applying a hybrid of dynamic and static analysis, the probability of correctly detecting malicious programs can be significantly increased. Testing their algorithm on 200 malware samples, the authors found that 33 % of malicious code analysed by the combined methods would not have been identified by dynamic analysis only.

Wagner and Dean [27] built non-deterministic push-down automata (NDPDA) accepting valid sequences of APIs, obtained through static analysis of the source code. At execution time this automata is used for evaluating the runtime sequence of API calls. If a runtime sequence of APIs is not accepted by automata, that might signal an intrusion. Another approach which utilises both static and dynamic analysis in order to find detect injected, dynamically generated and obfuscated code is introduced by [20]. In fact, this approach proposes an anomaly based technique where static analysis is assisted by dynamic analysis. It is used to identify the location of API calls within the program. The programs can be dynamically monitored later to verify that each observed system call is made from the same location identified using the static analysis. These both methods are promising; however, they need to perform dynamic analysis in order to offer a perfect detection. Therefore, this matter leads them to suffer from heavy processes of executing programs.

In order to rise the detection speed, it is essential to avoid using dynamic analysis in the scanning phase (i.e. test phase). There are some approaches which attempt not to utilise dynamic analysis in their procedures or at least try to perform it prior to start the scanning process. A method attempting to analyse an executable file without execution is proposed by Bergeron et al. [3]. In this approach the given executable is first disassembled; then, the assembly code is parsed to generate a syntax tree from its control flow graph (CFG) which is created before. An API-graph is constructed from this CFG where only API calls are preserved while all other assembly instructions are not involved. A critical-API graph is created from this API-graph where the user determines what APIs are critical through the use of security policies which are represented as an automaton, also referred to as security automaton.

One further approaches which extracts API calls via static analysis is "Static Analysis for Vicious Executables (SAVE)" [24]. It implements signatures in the form of API calls. In this method, first, the binary file is decompressed (if it was packed), then it is parsed in order to extract the API call sequence. Then, the API calls are compared against the signature calls using a similarity measure. If the similarity was above certain threshold, then the test file is identified as a known malicious program. Although this method works properly on the given sample set, it has some weak points. Most of the test samples were worms. Furthermore, the technique is mostly prone to false-positive errors, as API calling sequence can have high similarity between different malware as they have similar behaviour.

A pleasant collaboration between static and dynamic analysis techniques can reduce their drawbacks. Using dynamic analysis information on the under inspection program's points which are ambiguous for static analysis makes it more precise than before. Developing such a system is an incredibly hard task which the system must manage dynamic analysis on an executable file while the static analysis analyses its codes. It needs some braking points being identifiable by both dynamic and static analysis techniques. These break points are widely used by developers in order to debug their programs which is not available in the most released version of software particularly in malware. This paper proposed the API calls as break point which is approachable by dynamic and static analysis in the vast majority of executables. Since it is required break points being on decision making statements, API calls are not perfect for this task. Thus, the presented approach, HDM-Analyser, make use of some statistical information which is obtained by performing a matching algorithm on the situations of API calls in the source code and their position in the sequence of called APIs at runtime. Although HDM-Analyser does not extract the sequence of called APIs as exactly as dynamic analysis does, its analysis speed is incredibly higher rather than dynamic method.

3 Proposed method: HDM-Analyser

Applying data mining techniques on API call sequences is a popular and effective approach to detect unknown malware. In fact, an API call sequence can represent the behaviour of a program; therefore, it is an appropriate feature which can be used to classify unknown executable files according to their behaviour. Typically, an API call sequence is extracted by analysing the under inspected executable. As mentioned before, malware analysis methods are categorised to static, dynamic, and hybrid analyses. Static analysis investigates the source codes of the given executables in order to provide some information such as API calls, assembly instructions etc. which are able to describe them. On the

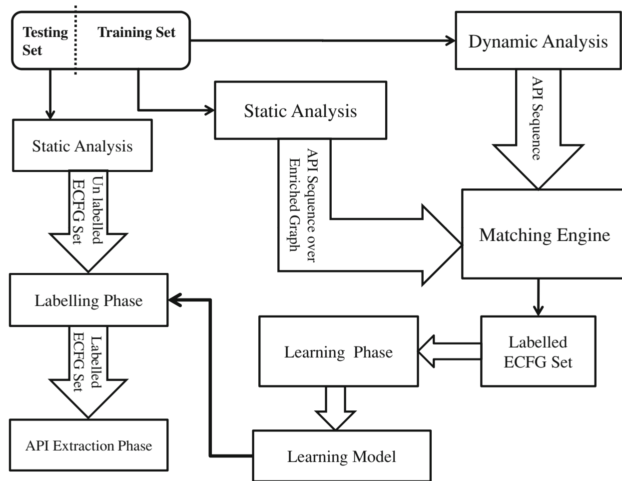


Fig. 1 An overall view of HDM-Analyser. It makes a use of a learning model in order to overcome the ambiguity of static analysis in conditional jump statements. This learning model, which is constructed by combining the outcomes of static and dynamic analysis methods, helps static analysis to choose the way which would be selected if the program will be executed. This type of selection is not based on guess and works for most of the times properly

other hand, dynamic analysis executes the input file and monitors its behaviour to capture required information. This type of analysis mostly logs the called APIs. There are some difficulties with both of considered analysis techniques. Although static analysis is quite faster than dynamic, it has a problem with conditional statements in the input code. The major issues of dynamic analysis are execution overhead and risky environment which needs to be isolated to prevent damaging the system. Hybrid analysis combines both static and dynamic analyses to perform a perfect analyse on executable files. This paper presents a novel approach that utilises machine learning techniques with taking advantages of considered analysis methods in order to improve the accuracy of malware analysis procedure while preserving its speed at a point reasonable. We call this approach as HDM-Analyser that stands for a Hybrid Analyser based on Data Mining techniques.

An overall view of HDM-Analyser is outlined in Fig. 1. Since this system needs a sequence of called APIs for each executable file which gathered by dynamic analysis, API call sequence of input file is extracted by dynamic analysis module; meanwhile, static analysis is performed on the input file in order to construct an ECFG model incorporated with API calls. ECFG is an Enriched Control Flow Graph representing control flow structure of a program. In fact, it gives more information about the program than classic CFG. A control flow graph, in short terms CFG, is a directed graph in which each node represents any kind of jump instruction or procedure call in the code sequence and each edge point to the jump destination. ECFG and its incorporation with API calls are described in Sect. 3.2. The matching engine combines the information obtained by dynamic analysis with corresponding ECFG; and then, each conditional jump receives a label according to dynamic information. At this point, a machine learning algorithm is employed to build a learning model with the labelled nodes of ECFG. This learning model will be used by HDM-Analyser at scanning time for analysing unknown executable files.

3.1 Normalising original assembly codes

HDM-Analyser applies a code normalisation on the original assembly codes in order to obtain a canonical form. This form is simpler than the original code in terms of set of instructions and syntax with a reasonable generalisation level. For instance, there are various kinds of jumps in assembly code while in the normalised version there are solely three jump instructions which are mapped to three different groups of jump statements. In addition, stack elimination is used to clarify the assembly code by replacing stack instructions with `MOV` statement. Furthermore, the operators with similar mechanism and final effects are unified by mapping to a single instruction, for example, any kind of addition or subtraction are mapped to `ADD`, multiplications and divisions are mapped to `MUL` and shifts are unified. The given assembly file is abstracted as a small set of notations. Table 1 introduces this abstraction mapping.

Procedure call and return instructions have a different story; therefore, it needs to take another method to make them normalised. That is why that `CALL` and `RET` instructions are not mentioned in Table 1. Because of their nature, all procedure calls and return instructions should be transformed to unconditional jumps and it is managed by Algorithm 1. There is one further call instruction which is used to invoke API functions. In the normalisation process, an API call instruction is mapped to `API <number>` notation, where this number corresponds the *API-Id*. An API-Id is an integer number that mapped to an API to facilitate the whole process.

3.2 Creating an intermediate representation by incorporating called APIs in ECFG model

When the normalised code is available, the enriched CFG construction can be followed.

The term "Enriched" refers to a node having a significant information about its neighbour codes. As mentioned before, a CFG is an abstraction model for the program's structure. The first pace of CFG construction is traversing normalised assembly code. Starting from the first line of the program, the node construction will continue till any type of jump is met. According to conditions of the jump, any possible happening will lead to a new node through an edge; hence, every node begins to form when a jump occurs. This trend makes

Table 1 Instruction mapping table

Assembly instructions	Normalised notations
JE, JZ, JNE, JNZ, LOOP, LOOPZ, LOOPE, LOOPNZ, LOOPNE, JCXZ	JE
JA, JG, JAE, JGE, JNB, JNL, JNBE, JNLE, JB, JL, JLE, JBE, JNG, JNA, JNGE, JNLE, JO, JC, JNO, JNC, JP, JNP	JC
JMP	JMP
ADD, SUB, INC, DEC, CMP, NEG	ADD
MUL, IMUL, DIV, IDIV, SHR, SAR, SHL, SAL	MUL
TEST, AND, OR, NOT, XOR	AND
ROR, ROL	ROT
PUSH, POP, MOV, LEA, MOVZX	MOV
All register operands	REG
All variable operands	VAR

The normalisation algorithm makes use of this mapping table to normalise instructions of the given assembly code

Algorithm 1: CnvProcCallToJmp($code$, $Line$)

```
Input: code is the input assembly code,
       Line is an object which contains some information about
       the line that procedure is called.
1  CalledLine = Line;
2  tmpJmp = new JMP instruction;
3  tmpJmp.target = Line.instruction.addressOfProc;
4  Line.instruction = tmpJmp;
5  while ( typeof( Line.instruction) != ProcEnd) do
6      if ( typeof( Line.instruction) == ProcCall) then
7          // Handle nested procedure calls
8          CnvProcCallToJmp( code, Line);
9          code.add( copyOfProc( Line));
10     end
11     if ( typeof( Line.instruction) == ProcReturn) then
12         tmpJmp = new JMP instruction;
13         tmpJmp.target = CalledLine.number + 1;
14         Line.instruction = tmpJmp;
15     end
16     Line.number++;
17 end
```

a simple control flow graph. To enrich this graph, an inspection phase evaluates the usage of frequency for normalised operations from certain number of instructions before current instruction and the final results are saved in the current node. The number of instructions taken into account for analysis is known as *Enrichment Factor*, denoted by ξ through the paper. ECFG construction is represented by Algorithm 2. This algorithm receives the following inputs: ξ (enrichment factor) and the CFG, and then enriches the input CFG. This algorithm moves a window of size ξ on the input CFG and stores the statistical information of the instructions within the window into the branch nodes as the enrichment vector. Once

Algorithm 2: enrichCFG(ξ , CFG)

```
Input: ξ is the enrichment factor (window size),
       CFG is the input control flow graph.
Output: An enriched control flow graph.
1  ECFG = CFG;
2  foreach ( node in ECFG);
3  do
4      List = A list of ξ predecessor nodes;
5      EV = new empty vector; // Enrichment vector
6      foreach ( item in List);
7      do
8          EV[ item.instType ]++;
9          EV[ item.fistOperandType ]++;
10         EV[ item.secondOperandType ]++;
11     end
12     node.EV = EV;
13 end
14 return ECFG;
```

the algorithm has built the $ECFG$ using the given inputs, the ECFG model is constructed.

According to the amount of enrichment factor, the algorithm saves the normalised operators' frequencies of ξ earlier instructions in the current node. Due to the importance of the API calls, the associated number of called API is kept in each node. An example of two enriched nodes is depicted in Fig. 2. Here $\xi = 3$ and two nodes are enriched with their three previous instruction logs. The optimum value for ξ is obtained by experiment. It is worth mentioning that each API call is placed on the edge which the API is called between its start node and its final node. HDM-Analyser uses these APIs as break points to match the information gathered by static and dynamic analyses. This matching is like debugging an under development software. This trend is described in next sections of the paper.

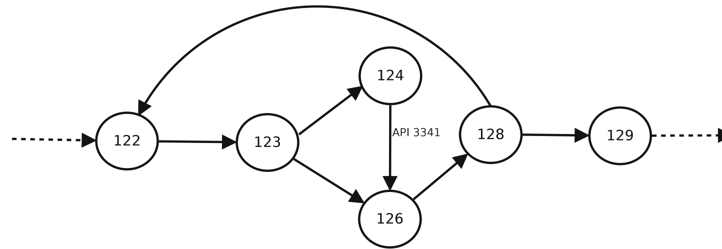
An interesting question that might arise here is that in Fig. 2, the statement in line 126 is assigned a specific node in CFG, whereas, there exists no node for line 127. As a matter of fact, a node in CFG representation scheme depicts either a jump instruction or that of its target. According to this definition, line 126 in Fig. 2 needs to be shown in CFG, however, this is not the case for line 127.

3.3 Building learning model based on hybrid analysis

As mentioned before, static analysis has a drawback in handling conditional jump statements. In the classic static analysis the decisions are made randomly because there is no knowledge about each alternatives. HDM-Analyser constructs a learning model in order to describe the behaviour of branch instructions of a program at runtime; and then, it offers the correct alternative on conditional jumps with a reasonable accuracy. Each branch behaviour is indicated with a positive or a negative label. A positive label signifies the

```
120: AND  REG, REG
121: ADD  VAR, REG
122: ADD  REG, REG
123: JC   126
124: ROT  REG
125: API  3341
126: MOV  REG, VAR
127: MOV  VAR, REG
128: JE   122
129: MUL  REG
```

(a)



(b)

	JE	JC	JMP	ADD	MUL	AND	ROT	MOV	REG	VAR
***Node*$_{123}$**	0	1	0	2	0	1	0	0	5	1
***Node*$_{128}$**	1	1	0	1	0	0	0	2	4	2

(c)

Fig. 2 An example of two enriched nodes of the ECFG. In this example, ξ is set to 3; therefore, each node in ECFG is enriched with statistical information extracted from 3 instructions prior to the statement in question. The best value for ξ will be obtained by experiment and the current value is just an example to show how the system works. **a** A set of normalised codes. **b** Its graphical model. **c** Enrichment vector of this set of codes in ECFG model

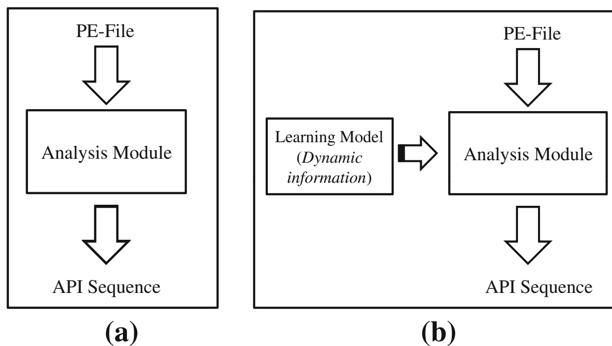


Fig. 3 Extracting API call sequence from a PE-file. HDM-Analyser augments the static analysis by making use of a learning model on ambiguous points of code. In fact, it extracts the API call sequence of an executable file by a kind of static analysis instead of executing it; thus, it is safer and faster than dynamic analysis. **a** Dynamic analysis. **b** HDM-Analyser

branch statement has jumped at runtime and a negative label shows that there was no jumping at this point. According to Fig. 3b, a learning model, which is constructed in the prior phases, is utilised to extract API sequence from a PE-file (i.e. portable executable file). This API sequence is similar to the API sequence acquired by dynamic analysis for the same input PE-file.

Since HDM-Analyser uses a learning model for predict the correct alternative on decision making points, in order to have ability of analysing unseen executable files, it has to be trained by known samples in advance. Training set is composed of considerable number of data-items. Each data-item is a conditional jump with its own label. In fact, a data-item in HDM-Analyser is an enrichment vector which contains some statistical information about normalised statements and a positive or negative label. Each executable file might have many enrichment vectors; therefore, there must be lots of data-items in the training set.

The labelling procedure is handled by Algorithm 3. This algorithm takes two parameters which are F, S. F is the first node of unlabelled ECFG graph which constructed out of an executable file codes and S is the API call sequence obtained by dynamic analysis for that file. i is a parameter which is used by algorithm in order to traverse the graph recursively. This algorithm traverses the given ECFG and investigates all enrichment vectors situated in it; and then, by making use of API call sequence, which is generated by dynamic analysis, specifies the label of each enrichment vector. Note that if the algorithm visits an empty edge, without any API on it, ignores this edge and goes to next node.

There are two known limitations for HDM-Analyser. The first one is in training set construction process. There are some branch nodes in ECFG model that might have no API on their edges. This matter means in some conditional statements such as *loops* and *if* instructions, there is no API which is called at runtime. Therefore, according to Algorithm 3 it is not possible to consider a label for such a node. Since the nodes in this situation do not receive any label, they are removed from enrichment table; thus, if the number of samples would be considerably large, this kind of node has no impact on efficiency of the training set. Figure 4 depicts an example of this type of branch node, which is highlighted by a question mark. On further restriction of HDM-Analyser is dealing with jumps which their targets are computed at runtime. It is impossible or quite hard to specify their targets by static analysis; hence, HDM-Analyser ignores them. Given the fact that this limitation belongs to CFG model, HDM-Analyser inherited it by CFG in static analysis phase.

Algorithm 3: SetLabel(F, S, $i = 0$)

Input: F is first node of an ECFG model constructed for a PE-file,
S is API call sequence of same PE-file extracted by dynamic analysis,
i an Integer variable for accessing to S cells.
Output: A labelled ECFG model.

```
1  //F.nextEdge is an edge pointing to next node of F
2  //F.jumpEdge is an edge pointing to target of jump instruction
   in F
3  if ( F.nextEdge.API == S[i]);
4  then
5  |  F.label = '-'; // Negative label
6  |  SetLabel( F.nextEdge.targetNode, S, i + 1);
7  end
8  if ( F.jumpEdge is exist and F.jumpEdge.API == S[i]);
9  then
10 |  F.label = '+'; // Positive label
11 |  SetLabel( F.jumpEdge.targetNode, S, i + 1);
12 end
13 if ( F.nextEdge.API is not exist);
14 then
15 |  SetLabel( F.nextEdge.targetNode, S, i);
16 end
17 if ( F.jumpEdge.API is not exist);
18 then
19 |  SetLabel( F.jumpEdge.targetNode, S, i);
20 end
```

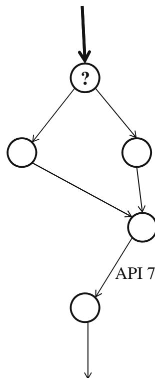


Fig. 4 An example of HDM-Analyser limitation. This approach is not able to realise the label of nodes with no API calls on their edges like the node indicated with a question mark in this figure. These nodes are ignored in learning phase and because of data-items' plurality, these nodes do not have bad impacts on analysis process

As Fig. 5 shows, there is a set of normalised assembly codes for N input PE-files, which is denoted by $C = \{c_1, c_2, \ldots, c_N\}$. The set of ECFG models, built by C set, is indicated with $G = \{g_1, g_2, \ldots, g_N\}$. The nodes of all G members receive their labels according to Eq. (1). All labelled ECFG graphs, which their nodes have received their labels, are put in a set, indicated by $GL = \{gl_1, gl_2, \ldots, gl_N\}$.

$$L(v_k) = \begin{cases} + & \text{if } v_k \text{ jumped at runtime} \\ - & \text{if } v_k \text{ did not jump at runtime} \end{cases} \tag{1}$$

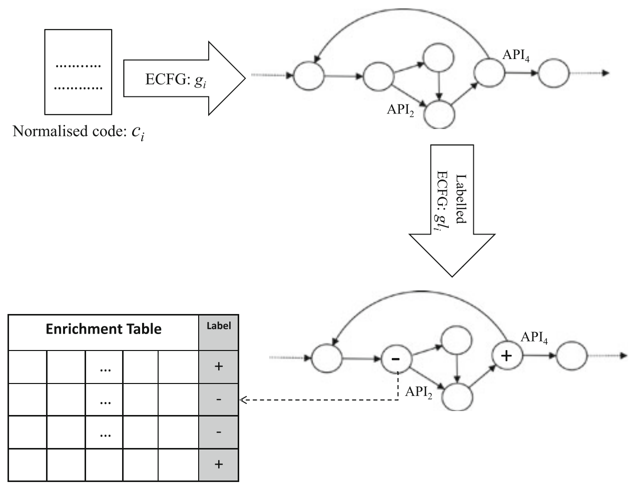


Fig. 5 Training set construction with a sample data. An ECFG representation model, g_i is created with a normalised code, C_i. Label of each branch node of g_i is resolved and put on it by Algorithm 3. This algorithm traverses the given ECFG and investigates all branches; and then, by utilising of API call sequence, extracted by dynamic analysis, specifies the label of each conditional jump node. gl_i notation is used for g_i which is received its labels. Finally, enrichment table is constructed by traversing gl_i. Each row of this table with its label indicates a labelled vertex in ECFG. HDM-Analyser uses a set of enrichment tables to collect the training set

where v_k indicates the kth vertex in g_i. An enrichment table, et, is a table which stores all of enrichment vectors of a labelled ECFG graph like gl_i. In other words, each row in et_i can be mapped to a branch node in gl_i. All enrichment tables are collected in this set: $ET = \{et_1, et_2, \ldots, et_N\}$. Training set is constructed by merging the members of ET. which is shown by $T = \{d_1, d_2, \ldots, d_k\}$, where d_i denotes a labelled enrichment vector of a graph and k is acquired by Eq. (2).

$$K = \sum_{i=1}^{N} |et_i| \tag{2}$$

where N is the number of ECFGs. Figure 6 illustrates a graphical view of the training set.

Testing set is a set which is used for evaluating the proposed approach in terms of accuracy and time. Each testing item can be considered as an unknown executable file in the wild. An unknown executable has to be formed as an understandable model for HDM-Analyser. Testing set construction is as exactly same as training set preparation with an exception in labelling procedure. As a matter of fact, a testing item has no label and the system predicts its label, by utilising a classifier, in testing phase. Once the testing set items are classified, their received labels are used to measure performance of the system.

Since HDM-Analyser is focused on speed with reasonable accuracy, it needs a learning model with low complexity. One of the learning models which satisfies both of these aims is Bayesian network.

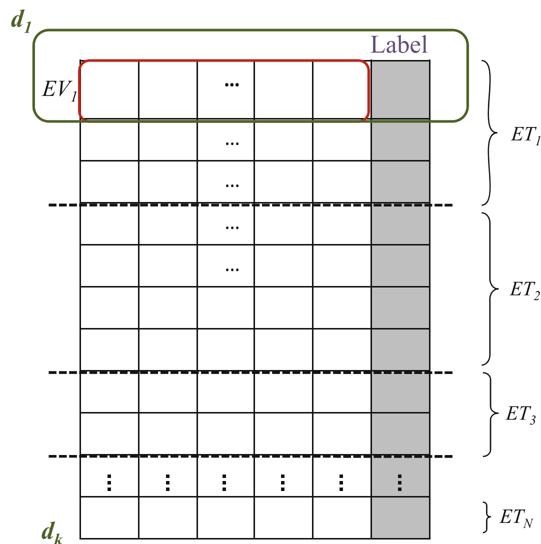


Fig. 6 Training set in tabular form. Each *row* of this table represents an enrichment vector which is constructed by collaborating static analysis with dynamic one on a specific executable file. Training set consists of all enrichment tables extracted from all ECFGs

3.4 Bayesian network as classification module of HDM-Analyser

One of the most efficient classifiers, in terms of predictive performance is competitive with state-of-the-art classifiers, is naive Bayesian classifier [8,14]. This classifier is trained the conditional probability of each attribute A_i given the class label C by make use of training data. Classification task is then done by applying Bayes rule to compute the probability of C given the particular instance of $A_1, \ldots, A_n$; and then predicting the class label with the highest posterior probability. This computation is rendered feasible by making a strong independence assumption: all the attributes A_i are conditionally independent which are given the value of the class C. This independency means probabilistic independence, that is, A is independent of B given C whenever $Pr(A|B, C) = Pr(A|C)$ for all possible values of A, B and C, whenever $Pr(C) > 0$. Since this assumption is clearly unrealistic, the performance of naive Bayes is somewhat surprising. In real world issues, the attributes are depended to each other.

In order to overcome the dependency problem, an appropriate language and efficient machinery to represent and manipulate independence assertions is required which both are provided by Bayesian networks [17]. These networks are directed acyclic graphs which allow efficient and effective representation of the joint probability distribution over a set of random variables.

Each node is associated with a probability function that takes a particular set of values as input for the node's parent variables and gives the probability of the variable represented by the node. This function is illustrated in Eq. (3). Let $G = (V, E)$ be a directed acyclic graph where V stands for vertices and E for edges and let $X = (X_v)$, $v \in V$ be a set of random variables indexed by V, then X is a Bayesian network with respect to G if its joint probability density function (with respect to a product measure) can be written as a product of the individual density functions, conditioned on their parent variables.

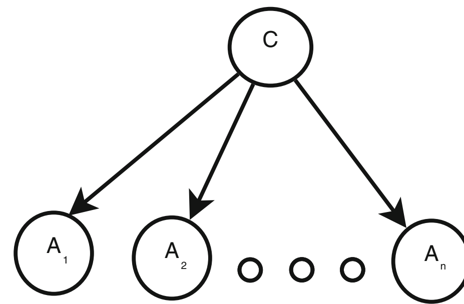


Fig. 7 Structure of naïve Bayes network

$$p(x) = \prod_{v \in V} p(x_v | x_{pa(v)}) \tag{3}$$

where $x_{pa(v)}$ is the set of parents of v (i.e. those vertices pointing directly to v via a single edge). Typically, node's predecessors are referred as parents in the graph.

In addition, the network encodes the following conditional independence statements: each variable is independent of its non-descendants in the graph given the state of its parents. These independencies are then exploited to reduce the number of parameters needed to characterise a probability distribution, and to efficiently compute posterior probabilities given evidence. Probabilistic parameters are encoded in a set of tables, one for each variable, in the form of local conditional distributions of a variable given its parents. Using the independence statements encoded in the network, the joint distribution is uniquely determined by these local conditional distributions [10].

If represented as a Bayesian network, a naive Bayesian classifier has the simple structure depicted in Fig. 7. This network captures the main assumption behind the naive Bayesian classifier, namely, that every attribute (every leaf in the network) is independent from the rest of the attributes, given the state of the class variable (the root in the network).

HDM-Analyser builds the learning model as the form of a Bayesian Network. This model is used to clarify the branch nodes of new ECFGs, extracted from new executables, for static analysis module. By utilising this learning model, the static analysis module is able to choose the correct way in conditional statements with a desirable level of satisfactory. Once the execution path is specified, the system is simply able to extract the API call sequence by traversing this path.

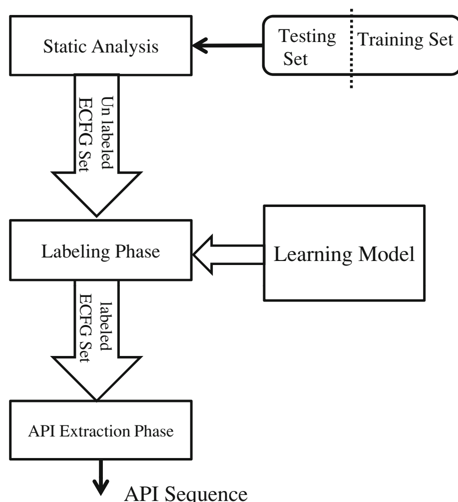


Fig. 8 API call sequence extraction. HDM-Analyser utilises a learning model in order to predict the labels of each branch node of the input executable. Once their labels are realised, the system knows that which branches will be jumped if the program is executed; thus, it is able to follow a reasonable approximation path of the real execution path and extract the API calls. This path is highlighted on each ECFG model; and then, the system traverses the given ECFG through this path in order to find the API calls. This trend is shown in Algorithm 4

Algorithm 4: extractAPISequence(F)

Input: F is the first node of an ECFG model constructed for a PE-file.
Output: S is an API call sequence of the same PE-file.

1 S is an empty set to store the API calls.
2 //$F.nextEdge$ is an edge pointing to the next node of F
3 //$F.jumpEdge$ is an edge pointing to the target of jump statement in F
4 **if** ($F.label$ *is exist* ***and*** $F.label$ == '+') **then**
5 $S \leftarrow S \cup F.jumpEdge.API$;
6 return $S \cup$ extractAPISequence($F.jumpEdge.targetNode$);
7 **else**
8 $S \leftarrow S \cup F.nextEdge.API$;
9 return $S \cup$ extractAPISequence($F.nextEdge.targetNode$);
10 **end**
11 return S;

3.5 Extracting API sequences by HDM-Analyser

Extracting an API call sequence from an ECFG model needs to traverse it from its first node to end node on correct execution path, which is followed by a program at its execution time. HDM-Analyser by making use of a learning model and a classification algorithm is able to predict this path for a given input executable file without running it. Figure 8 depicts overall structure of the API sequence extraction procedure.

Each branch node in a new ECFG receives its label from the classification module. This process is done for all branch nodes of the ECFG in labelling phase; following that by utilising Algorithm 4 the API call sequence is extracted. This algorithm takes the first node of an ECFG and traverses it recursively in order to pick up the API calls which are situated on edges in its path. When this algorithm faced to a branch node, it chooses the correct way according to the node's label. If the label is positive, it realises that the program will be jumped at this point if it is executed; therefore, it takes the jumping edge to continue its way. On the other hand, if the edge have no label or its label is negative, the algorithm simply ignores it and continue the previous trend.

4 Experimental setup

Ability of extracting the real sequence of API calls is a pivotal matter for malware detection systems which their detection procedure are working based on mining API calls. These systems are different in terms of accuracy and detection speed. As matter of fact, dynamic analysis is more accurate than static analysis in extracting API calls sequence; however, static analysis is faster and safer than dynamic one because it does not need to execute executable files. Static analysis is able to offer an approximation sequence of the actual API call sequence; in contrast, dynamic analysis extracts the real API call sequence. This is a trade-off between speed and accuracy. The presented approach, HDM-Analyser, attempts to raise both of them simultaneously by combining static and dynamic analysis. HDM-Analyser takes advantages of both analysis methods.

In order to evaluate HDM-Analyser capabilities, two categories of experiments are performed. The first one compares HDM-Analyser with static and dynamic analysis methods in terms of speed and accuracy. This comparison is done by matching API call sequences extracted by these analysis techniques with each other. The next category of experiments makes use of the extracted API call sequences in a data mining based detection technique in order to measure the detection accuracy in realising unknown malware. As this paper is focused on malware analysis, the first experiment is more important than the second one.

In the following, the dataset which is used in the experiments is described briefly.

4.1 Experimental results for evaluating the analysis ability of HDM-Analyser

Since HDM-Analyser is proposed for malware analysis, it is vital to evaluate its analysis ability. This evaluation is done

by calculating the similarity of API call sequences generated by each approach with real API call sequences. This experiment is performed on a set of executable files which are analysed by static analysis, dynamic analysis, and HDM-Analyser. Some times API call sequences extracted by several analysis methods are different and not aligned respect to each other; therefore, it is essential to have an algorithm to measure the similarity of these sequences. This algorithm is presented in this section.

4.1.1 Dataset description

Since this study is focused on analysing PE-files, this experiment is carried out on 1,000 Windows based 32-bits network worms. These portable executable malware samples are selected randomly from malware repository of APA, the security research laboratory at Shiraz University. In fact, this repository has been collecting since 2010 and updating with new malware instances daily. Each day over 3,000 executable malicious files are added to this repository from online malware repositories such as VirusSign (http://www.virussign.com). VirusSign offers a huge collection of high quality samples, it is a valuable resource for anti-virus industry, it has opened the samples to help security corporations to improve their products.

4.1.2 Performance measure

Since the real API call sequence is extracted by dynamic analysis, the output sequence of this analysis method can be used as a criterion for evaluating the accuracy of other API call extraction techniques. Thus, in order to evaluate the accuracy of HDM-Analyser, it is required to measure similarity between API sequences extracted by this approach and dynamic analysis.

Typically, various API call extraction approaches produce different API call sequences for a specific PE-file. These approaches sometimes miss some API calls or put some APIs in their output sequence while the program never call those APIs in such positions in its real execution. Therefore, there are several sequences, extracted by these approaches, with different sizes and different alignments. In order to compare two extracted sequences of API calls, first, they have to be aligned to right position respect to each other; and then, computing the similarity measure between them become an easy task. A pleasant technique is employed to resolve this issue. Figure 9 illustrates an example of alignment matrix for two API call sequences. In fact, this matrix is a cross table in which the matched points of two sequences are marked in their corresponding cells.

Algorithm 5 utilises the alignment matrix in order to align the input sequences, and then calculates their similarity. This algorithm takes two sequences as input parameters which are S_E as the expected sequence and S_O as the observed sequence. At the first stage, the alignment matrix is constructed and stored in M_A. Next, the algorithm investigates all possible paths and give each path a specific score which is obtained by counting matches and mismatches. A path demonstrates that how many shifts in which direction are required in order to match S_O with S_E. Those shifts have to be continued to reach maximum match in two sequences. As a matter of fact, this is an optimisation problem; therefore, the algorithm calculates score of each path, and finally returns the maximum score as result.

Algorithm 5 computes the similarity value merely between two sequences. In order to compare two approaches, it

	API_1	API_2	API_5	API_7	API_9	API_{10}
API_1	x					
API_2		x				
API_7				x		
API_9					x	
API_{10}						x
API_8						

Fig. 9 An example of alignment matrix. This *matrix* demonstrates the matched points of these sequences. Alignment matrix is employed by Algorithm 5 in order to compare two given API call sequences with each other

Algorithm 5: SeqSim(S_E, S_O)

Input: S_E is the Expected sequence;
S_O is the Observed sequence;
Output: The similarity of two sequences

```
1  S_PathScore is an empty vector;
2  M_A = Construct the alignment matrix for S_E and S_O;
3  foreach element i in S_O do
4  |   S_PathScore[i] = 0;
5  |   S_tmp = S_E;
6  |   foreach element k in S_O do
7  |   |   p = Find position of k in S_tmp by using M_A;
8  |   |   if (p == 0 /* Not found */);
9  |   |   then
10 |   |   |   S_PathScore[i] += -1/2;
11 |   |   |   continue;
12 |   |   end
13 |   |   S_PathScore[i] += 1 - log(p);
14 |   |   remove elements of S_tmp from index 0 to p;
15 |   end
16 |   remove element i from S_O;
17 end
18 bestPathScore = max(S_PathScore);
19 return bestPathScore / |S_E|;
```

needs to calculate similarity value amongst all corresponding sequences extracted by them. Hence, an average of these similarities is used, which is obtained by Eq. (4).

$$AvgSim(M_{SE}, M_{SO}) = \frac{1}{N}\sum_{i=1}^{N} SeqSim(S_{E_i}, S_{O_i}) \tag{4}$$

where M_{SE} is a set of expected API call sequences extracted from a set of input PE-files. M_{SO} is observed API call sequence set which is extracted by another approach from same set of input executable files. In this paper, expected sequences are produced by dynamic analysis technique and observed sequences are extracted by considered approach such as static analysis and HDM-Analyser. The total number of input files is denoted by N.

As mentioned ago, the speed of analysis process is another important evaluation measure. Obviously, running a program in order to analyse its behaviour takes considerable execution time. This time consumption is related to input program's codes. A program with a few instructions may take more execution time than a longer one. Typically, high iterative loops are responsible for heavy execution overheads. Obviously, several programs which provide single functionality written by various developers can be dissimilar in code structure and compiler. Therefore, they have different time complexities and it is impossible to ignore effects of programmer coding style and compiler while comparing the dynamic analysis by make use of time complexity equations. According to this difficulty, one of the interesting alternatives is using the elapsed time as measure. Thus, we employ this measure to evaluate the analysis speed of the presented approach.

4.1.3 Discussion

Accuracy and speed of analysis is measured to compare performance of rival analysis approaches. Since higher accuracy leads to have a powerful analysis ability, malware detection methods based on better analysis become more accurate. A malware detection system provides a safe environment for user's programs and data. It is essential to provide the security while do not imposing a heavy overhead on the system. Thus, providing an accurate and fast malware analysis system, which is used in malware detection systems, is the goal of HDM-Analyser.

When a program is executed, it requests its required services by calling APIs. Monitoring a program at runtime and picking up the called APIs generates a realistic API sequence. This procedure is followed by dynamic analysis; therefore, the output of this method can be used as a benchmark for other analysis techniques in terms of accuracy. In this experiment, *API Logger* tool is employed to extract a sequence of called API's from a PE-file for dynamic analysis part. API Logger is able to log any exported API's and display wide range of information, including API name, call sequence and more [9].

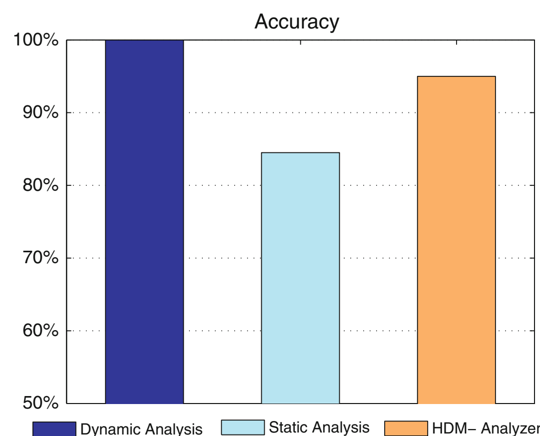


Fig. 10 Accuracy comparison in API sequence extraction amongst static, dynamic analysis and HDM-Analyser. According to this experiment, HDM-Analyser has achieved a higher accuracy than static analysis method, which are 95.27 and 89.43 % respectively, and has obtained a slightly lower accuracy than dynamic technique

Figure 10 indicates some results used for comparing the accuracy of various analysis approaches with each other. Since dynamic analysis is considered as benchmark, it has the highest accuracy rate in this chart. According to this results, static analysis obtains the lowest accuracy rate against its rival approaches in this experiment. This approach, makes random decisions to select its path in branch nodes in traversing control flow graph; therefore, it produces an API sequence which is much differ from the benchmark API call sequence; and consequently, it looses its accuracy. In this experiment, HDM-Analyser has achieved a higher accuracy than static analysis method and has obtained a slightly lower accuracy than dynamic technique.

An interesting question might arise here is that, in Fig. 10, since static analysis uses random choice in branch nodes, it seems its accuracy must be approximately 50 or 60 %, while it has significantly gained higher value than this amount for accuracy measure. As a matter of fact, in a program all of the instructions are not involved in decision making statements which have impacts on API call sequence. In addition, there are some other decision making statements such as "loops" or "if" instruction without any API call occurrence within them. Therefore, the accuracy of API call sequence extraction can be affected merely by "if" statements which have at least one API call in their code bocks.

As mentioned ago, HDM-Analyser is based on a hybrid analysis technique which combines the information gathered by static and dynamic analyses. A learning model is constructed by API call sequences that extracted by dynamic analysis and a set of ECFG models generated by static analysis. This learning model guides the system to make a better

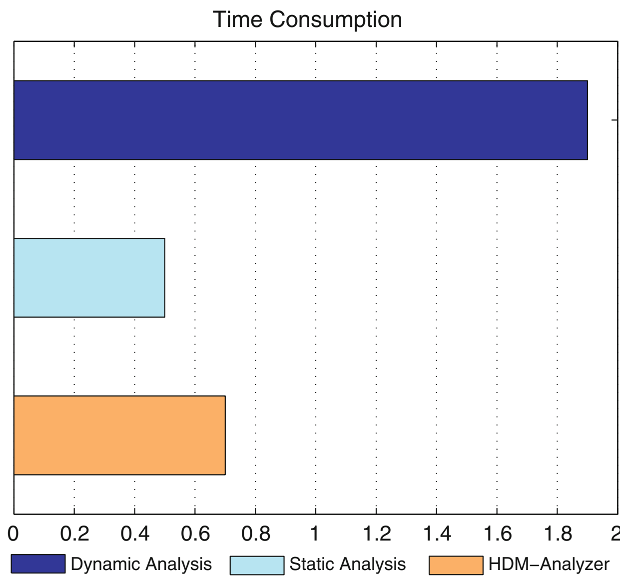


Fig. 11 A comparison among static, dynamic analysis and HDM-Analyser in terms of analysis speed. These results show an average time consumption in seconds for analysing an executable file which analysed by each analysis method

decision in branch nodes. In other words, in HDM-Analyser, the static analysis section is augmented by dynamic analysis information. By employing this mechanism, the presented approach receives higher accuracy rate than static analysis method. In fact, decision making in HDM-Analyser is intelligent; however, in the static analysis its is mostly based on guess. By the way, the accuracy of the proposed approach is slightly less than dynamic approach, following from its limitations which are mentioned before; in contrast, it is quite faster and safer than dynamic analysis method.

Figure 11 illustrates a speed comparison amongst dynamic analysis, static analysis, and the presented approach. According to these experimental results, static analysis is the fastest approach. This speed is due to its algorithm which moves fast through the executable file's codes. It does not need to check anything or wait for some results during the process of its analysis procedure; therefore, it is the fastest analysis approach. In contrast, HDM-Analyser has to predict a class label for each branch node before making a decision about choosing the correct way; hence, it is slightly slower than static analysis. However, it has better accuracy than static analysis in API call extraction.

Nevertheless HDM-Analyser is slightly slower than static analysis, it achieves an incredibly accuracy rather than static one. This increase is a significant prosperity rather than decreasing a little speed in analysis process. On the other hand, its quickness cannot be compared with speed dynamic analysis. There is an unbelievable difference between their analysis speeds. Thus, generally HDM-Analyser has achieved better results than its rivals and it can be used instead of them in some applications; however, it has some limitations which should be resolved.

4.2 Experimental results for detection system

Since the major functionality of the presented approach is extracting API call sequences from programs, it can be used as a part of feature extraction module in malware detection systems which are designed based on data mining techniques. There are incredible number of malware detection systems which make use of API calls as an effective feature in order to realise unknown malware [24,29,28]. Typically, these systems extract API calls from known executable files and put them in their training set. Next, they employ at least a classification algorithm and learn it over training set. Then, they use those classifier(s) to distinguish malware samples from sane ones. In this experiment several classifier are employed and same trend is followed to make a comparison between HDM-Analyser and traditional API extraction solutions in terms of accuracy. First of all, API call sequences are extracted by static, dynamic, and the presented analysis approach; and then, they are put in separated sets. Then each classifier is trained and evaluated on each set individually. In the following a brief description about the classifiers, which are used in the experiments, is explained.

A decision stump is a machine learning model consisting of a one-level decision tree [12]. It is a decision tree with one internal node (the root) which is immediately connected to the terminal nodes. A decision stump makes a prediction based on the value of just a single input feature.

Sequential minimal optimisation (SMO) is an algorithm for solving large quadratic programming (QP) optimisation problems, which is widely used for the training of support vector machines. SMO breaks up large QP problems into a series of smallest possible QP ones, which are then solved analytically [19].

A naïve Bayes classifier assumes that the presence (or absence) of a particular feature of a class is unrelated to the presence (or absence) of any other feature. It is a kind of dependent constraint modelling [15].

A random tree is a tree or arborescence that is formed by a stochastic process. There are different types of random trees including uniform spanning tree, random minimal spanning tree, random binary tree, random recursive tree, randomised binary search tree, rapidly exploring random tree, Brownian tree, random forest, and branching process [7].

K-Star is an instance-based classifier, which is the class of a test instance is based upon the class of those training instances similar to it, as determined by some similarity function. It differs from other instance-based learners in that it uses an entropy based distance function [6].

Random forest is an ensemble classifier that contains many decision trees and outputs the class that is the mode of the class's output by individual trees [5].

These classifiers are performed on the given dataset by *WEKA*. Weka workbench [11] contains a collection of visualisation tools and algorithms for data analysis and predictive modelling, together with graphical user interfaces for easy access to this functionality.

4.2.1 Dataset description

In order to prepare an experimental dataset, 2,000 executable files consisting of 1,000 benign programs and 1,000 malware are collected. Note that all samples are non-packed because this paper is not focused on packed executable files. Those benign programs are gathered from a fresh installed Microsoft Windows XP SP2 on a virtual machine. Moreover, some popular user applications are installed on that machine such as Microsoft Office, Adobe Photoshop, Music players and so on. As mentioned before, malware samples are selected randomly from malware repository of APA, the security research laboratory at Shiraz University.

4.2.2 Performance measure

In this section some typical measures, which are used for evaluating the detection accuracy of malware detection systems, are introduced. For classification tasks, the terms *true positive*, *true negative*, *false positive*, and *false negative* are utilised to make a comparison of different classifiers according to their results under test. The terms *positive* and *negative* refer to the classifier's prediction (usually known as the observation), and the terms *true* and *false* refer to whether that prediction corresponds to the external judgement (typically known as the expectation). Therefore, the term "True Positive" (TP) shows the number of correctly recognised items which were belonging to the goal set and "True Negative" (TN) indicates the number of correctly recognised items which were not belonging to the goal set. On the other hand, "False Positive" (FP) represents the number of incorrectly recognised item which were belonging to the goal set and "False Negative" (FN) illustrates the number of incorrectly recognised item which were not belonging to the goal set. We define "Detection Rate" as the percentage of all PE-files labelled "malicious" that can receive correct label by the system, as illustrated in Eq. (5).

$$DetectionRate = \frac{TP}{TP + FN} \tag{5}$$

"False Alarm Rate" is the percentage of labelled normal that likewise receive the wrong label by the system, as shown in Eq. (6).

$$FalseAlamRate = \frac{TP}{TP + FN} \tag{6}$$

"Accuracy" is the overall accuracy of the system in detection of malware and benign files, as illustrated in Eq. (7).

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{7}$$

To evaluate each classifier, 10-folded cross-validation method is used and the results are obtained over 10 independent runs of 10-folded cross-validation. "Cross validation", is a technique for assessing how the results of a statistical analysis will generalise to an independent data set. This technique is mainly used in settings where the goal is prediction, and one wants to estimate how accurately a predictive model will perform in practice. One round of cross validation involves partitioning a sample of data into complementary subsets, performing the analysis on one subset called the training set, and validating the analysis on the other subset called the testing set. To decline variability, multiple rounds of cross validation are performed using different partitions, and the validation results are averaged over the rounds, each round is called as a fold [18].

4.2.3 Discussion

As mentioned before, the objective of these experiments, the second section, is demonstrating the efficiency and usability of the API sequences which are extracted by HDM-Analyser. Since HDM-Analyser presents a new analysis approach, in this experiment, its application in detecting unknown malware is discussed. Obviously, one of the best features utilised by many data mining based detection approaches is API call. On the other hand, since a sequence of called APIs shows the program's behaviour, if the extraction procedure is done accurately, the detection accuracy will be soared. Thus, in order to show the API call extraction ability and accuracy of HDM-Analyser, the API call sequences extracted by HDM-Analyser and its rivals, static and dynamic analyses, are evaluated by same classifier and to increase the certainty this experiment is done on six classifiers.

Before employing a classifier to learn an evaluate the extracted features, it is essential to prepare them in an appropriate format. A popular form for representing data-items and their features is tabular form in which the features of each data-item are packed in a specific row and each column refers to a particular feature. In this form, the values of the features are written in the contributing cells. An example of this representation is illustrated in Fig. 12. In this table, each entity indicates the presence or absence of an API function called or not called by its relevant PE-file. This table is simply converted to a format being suitable for WEKA in order to be easy to use.

	API_1	API_2	API_3		API_m	Label
$PE\text{-}File_1$	0	1	1		0	+
$PE\text{-}File_2$	1	1	1		0	-
.						
.						
.						
$PE\text{-}File_N$	1	0	0		1	+

Fig. 12 A dataset consisted of both malware and benign executable samples. The features of this dataset are API calls where called or not called by each PE-file. Each row of this table depicts which APIs are called by PE-file$_i$. The value 0 indicates that such API is not called by PE-file$_i$ and 1 means that API is called by the ith PE-file. The last column is devoted to the label of each PE-file, positive value is referred to maliciousness and negative label shows the saneness of the PE-file

Figure 12 shows a dataset generated for classification purpose in order to evaluate the extracted feature set. A classifier is trained by training set and performs a prediction according to its learnings on testing set items. It classify these items by putting positive and negative labels on them which indicate that a sample is malware or benign respectively. Table 2 illustrates the detection accuracy amongst various analysis approaches for each introduced classifier. The results depicted in this table demonstrate that HDM-Analyser has achieved a desirable accuracy which is 92.68 % on average. It means it is suitable in order to be used as feature extractor module for malware detection systems which are based on data mining techniques. It is worth mentioning that in a malware detection system, the analysis speed is as important as the accuracy. In other words, gaining maximum accuracy of malware analysis imposes a heavy overhead on the system which causes a huge decline in speed, for instance, dynamic analysis has received maximum accuracy in these experiments which is 94.76 % on average, but it takes lots of time to analyse the input PE-files. Although HDM-Analyser is slightly less accurate than dynamic analysis method, it is very faster than this method according to earlier experimental results depicted in Fig. 11. Moreover, the results show that HDM-Analyser is more accurate than static analysis (88.57 % on average) which is discussed before.

The false alarm, false positive, rate is an effective measure that gives useful information in order to compare and measure the failure of variant classification operations. Table 3 illustrates a comparison of variant analysis methods per different classifier in terms of false alarm rate. False alarm rate of HDM-Analyser is lower than static analysis, 6.09 and 6.51 % on average respectively. Nonetheless dynamic analysis has the lowest false positive rate among the other approaches which is 5.91 % on average, it suffers from execution overhead which is discussed in the previous sections.

Tables 2 and 3 demonstrate that HDM-Analyser is not bias to a particular classifier as a malware detector. Thus, it can be used in variety systems with wide range of classifiers. One further fact which is revealed by these results is the effectiveness of the presented approach in analysing executable files. It indicates that HDM-Analyser can be accepted as an analysis approach for extracting API calls. According to reported results this approach has earned a pleasant performance than its rival approaches. Note that, in this situation, performance refers to both accuracy and speed of analysis and detection.

Table 2 Accuracy of malware detection ability among several analysis methods

	Dynamic (%)	Static (%)	HDM-Analyser (%)
Decision stump	89.76	83.89	87.79
SMO	93.01	86.93	90.97
Naive Bayes	92.47	86.42	90.44
Random tree	96.54	90.23	94.42
Lazy KStar	99.26	92.77	97.08
Random forest	97.53	91.15	95.39
Average	94.76	88.57	92.68

There are three datasets which generated by each analysis approach. Different classifiers perform classification on the executable files by making use of a same dataset. This trend is followed for each dataset. The evaluation results are depicted in this table separately. As it can be clearly seen, HDM-Analyser has attained an accuracy between the accuracy of dynamic and static analyses. These results demonstrate that HDM-Analyser is appropriate to be used as feature extractor module for malware detection systems which are based on data mining techniques

Table 3 The false alarm rate of a malware detection method which uses various analysis methods in order to extract the required features from executable files

	Dynamic (%)	Static (%)	HDM-Analyser (%)
Decision stump	8.53	9.47	8.85
SMO	8.01	8.90	8.32
Naive Bayes	8.43	9.17	8.55
Random tree	4.66	5.08	4.84
Lazy KStar	2.08	2.32	2.15
Random forest	3.72	4.12	3.87
Average	5.91	6.51	6.09

In this table, the false alarm rate of HDM-Analyser is compared with its rivals' results which are static and dynamic analyses. According to these results, HDM-Analyser has achieved to a lower false alarm rate rather than static analysis. This achievement is a consequence of using dynamic information in its analysing procedure in order to rise the accuracy

5 Conclusions and future work

API calls are important features being able to describe the behaviour of programs; therefore, they are appropriate features which can be used by a classifier in order to categorise executable files based on their behaviour. This categorisation

is widely used for detecting unknown malware. The feature set, which describes executable files, conducts a crucial role in detection; in other words, this is one of the most effective parameters being able to increase the detection accuracy. Typically, API calls are extracted by analysis methods, which investigate the codes of the under inspection programs or monitor their behaviour at runtime in order to find API calls. Thus, the performance of analysis method is a pivotal matter.

There are two major practical analysis techniques which are dynamic and static. Dynamic analysis, which executes the given executable file and extracts its API calls at runtime, suffers from execution overhead because it has to wait for the input file's responds. On the other hand, static analysis, which parses the codes of input file in order to extract needed information, does not have this problem, however its accuracy is lower than dynamic one in extracting API calls because it does not know which branch statement jumps at runtime.

In this paper, an approach is proposed which uses the advantages of both mentioned analysis techniques in order to reduce the effects of their drawbacks on analysis performance. This approach, HDM-Analyser, utilises the information obtained by dynamic analysis on static analysis ambiguous points and makes them clear for static analyser. In order to overcome the execution overhead, HDM-Analyser does the dynamic analysis on an enough large set of samples in advance and stores required information in form of learning model. This learning model is used by a classifier to illuminate the decision points, which are opaque for static analysis, and raise the static analysis accuracy. Since this approach uses both dynamic and static analysis techniques to extract API calls from executables, it can be involved into hybrid analysis approaches.

In order to evaluate HDM-Analyser abilities, two categories of experiments are performed. The first one compares HDM-Analyser with static and dynamic analysis methods in terms of analysis speed and accuracy. This comparison is done by matching API call sequences extracted by these analysis techniques with each other. According to this experiment, HDM-Analyser has achieved a higher accuracy than static analysis method, which are 95.27 and 89.43 % respectively, and has obtained a slightly lower accuracy than dynamic technique; in contrast, it is quite faster and safer than dynamic analysis method. In fact, this is a trade-off between speed and accuracy.

The next category of experiments makes use of the extracted API call sequences in a data mining based detection technique in order to measure the detection accuracy in realising unknown malware. The results demonstrate that HDM-Analyser has achieved a desirable accuracy which is 92.68 % on average. It means it is suitable in order to be used as feature extractor module for malware detection systems which are based on data mining techniques. Gaining maximum accuracy of malware analysis, extracting the exact APIs called at runtime, imposes a heavy overhead on the system which causes a huge decline in speed, for instance, dynamic analysis has received maximum accuracy in these experiments which is 94.76 % on average, but it takes lots of time to analyse the input PE-files. Although HDM-Analyser is slightly less accurate than dynamic analysis method, it is very faster than this method according to experimental results mentioned earlier. Moreover, the results show that HDM-Analyser is more accurate than static analysis (88.57 % on average).

This study opens new research directions to malware analysis methods in which incorporates data-mining techniques in order to boost the static analysis ability. We intend to expand this approach to handle the jump instructions relied on addresses which are computed at runtime. In addition, since some malware instances obfuscate their jump statements by replacing other instructions which do exactly these jumps' functionalities, it is a hard task to analyse them by current version of HDM-Analyser. These kinds of malware need different algorithms to analyse; thus, as future work, we are going to work on them.

References

1. Abou-Assaleh, T., Cercone, N., Keselj, V., Sweidan, R.: Detection of new malicious code using n-grams signatures. In: Proceedings of Second Annual Conference on Privacy, Security and Trust, pp. 193–196. Citeseer (2004)
2. Bayer, U., Kruegel, C., Kirda, E.: Ttanalyze: A tool for analyzing malware. In: 15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference. Citeseer (2006)
3. Bergeron, J., Debbabi, M., Desharnais, J., Erhioui, M., Lavoie, Y., Tawbi, N.: Static detection of malicious code in executable programs. Int. J. Req. Eng. **2001**, 184–189 (2001)
4. Bergeron, J., Debbabi, M., Desharnais, J., Ktari, B., Salois, M., Tawbi, N., Charpentier, R., Patry, M.: Detection of malicious code in cots software: A short survey. In: First International Software Assurance Certification Conference (ISACC99) (1999)
5. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
6. Cleary, J., Trigg, L.: K*: An instance-based learner using an entropic distance measure. In: Machine Learning-International Workshop Then Conference-, pp. 108–114. Citeseer (1995)
7. Dietterich, T.: An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. Mach. Learn. **40**(2), 139–157 (2000)
8. Duds, R., Hart, P.: Pattern Classification and Scene Analysis. Wiley, New York (1973)
9. Fasikhov, R.: Api logger tool. http://blackninja2000.narod.ru/rus/api_logger.html
10. Friedman, N., Geiger, D., Goldszmidt, M.: Bayesian network classifiers. Mach. Learn. **29**(2), 131–163 (1997)
11. Holmes, G., Donkin, A., Witten, I.: Weka: A machine learning workbench. In: Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on, pp. 357–361. IEEE (1994)
12. Iba, W., Langley, P.: Induction of one-level decision trees. In: Proceedings of the Ninth International Conference on, Machine Learning, pp. 233–240 (1992)

13. Idika, N., Mathur, A.: A survey of malware detection techniques. Purdue University (2007)
14. Langley, P., Iba, W., Thompson, K.: An analysis of bayesian classifiers. In: Proceedings of the National Conference on Artificial Intelligence, pp. 223–223. Wiley, Hoboken (1992)
15. Lewis, D.: Naive (bayes) at forty: The independence assumption in information retrieval. Machine Learning: ECML-98, pp. 4–15 (1998)
16. Orenstein, D.: Quickstudy: Application programming interface (api) (2000)
17. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann, Los Altos (1988)
18. Picard, R., Cook, R.: Cross-validation of regression models. J. Am. Stat. Assoc., 575–583 (1984)
19. Platt, J.: 12 fast training of support vector machines using sequential minimal, optimization (1998)
20. Rabek, J., Khazan, R., Lewandowski, S., Cunningham, R.: Detection of injected, dynamically generated, and obfuscated malicious code. In: Proceedings of the 2003 ACM workshop on Rapid malcode, pp. 76–82. ACM (2003)
21. Roundy, K., Miller, B.: Hybrid analysis and control of malware. In: Recent Advances in Intrusion Detection, pp. 317–338. Springer, Berlin (2010)
22. Sekar, R., Bowen, T., Segal, M.: On preventing intrusions by process behavior monitoring. In: USENIX Intrusion Detection, Workshop, vol. 1999 (1999)
23. Siddiqui, M.: Data mining methods for malware detection. ProQuest (2008)
24. Sung, A., Xu, J., Chavez, P., Mukkamala, S.: Static analyzer of vicious executables (save). In: Computer Security Applications Conference, 2004. 20th Annual, pp. 326–334. IEEE (2004)
25. Szor, P.: The Art of Computer Virus Research and Defense. Addison-Wesley Professional, Reading (2005)
26. Tzermias, Z., Sykiotakis, G., Polychronakis, M., Markatos, E.: Combining static and dynamic analysis for the detection of malicious documents. In: Proceedings of the Fourth European Workshop on System Security, p. 4. ACM (2011)
27. Wagner, D., Dean, R.: Intrusion detection via static analysis. In: Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on, pp. 156–168. IEEE (2001)
28. Xu, J., Sung, A., Chavez, P.: Mukkamala, S.: Polymorphic malicious executable scanner by api sequence analysis (2004)
29. Ye, Y., Wang, D., Li, T., Ye, D.: Imds: Intelligent malware detection system. In: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1043–1047. ACM (2007)