Computing and Informatics, Vol. 32, 2013, 1001–1030, V 2013-Feb-11

NOA: AN INFORMATION RETRIEVAL BASED MALWARE DETECTION SYSTEM

Igor Santos, Xabier Ugarte-Pedrero
Felix Brezo, Pablo G. Bringas

S^3Lab, DeustoTech – Computing
Deusto Institute of Technology, University of Deusto
Avenida de las Universidades 24, 48007, Bilbao, Spain
e-mail: {isantos, xabier.ugarte, felix.brezo, pablo.garcia.bringas}@deusto.es

José María Gómez-Hidalgo

Optenet, Madrid, Spain
e-mail: jgomez@optenet.com

Communicated by Deepak Gang

Abstract. Malware refers to any type of code written with the intention of harming a computer or network. The quantity of malware being produced is increasing every year and poses a serious global security threat. Hence, malware detection is a critical topic in computer security. Signature-based detection is the most widespread method used in commercial antivirus solutions. However, signature-based detection can detect malware only once the malicious executable has caused damage and has been conveniently registered and documented. Therefore, the signature-based method fails to detect obfuscated malware variants. In this paper, a new malware detection system is proposed based on information retrieval. For the representation of executables, the frequency of the appearance of opcode sequences is used. Through this architecture a malware detection system prototype is developed and evaluated in terms of performance, malware variant recall (false negative ratio) and false positive.

Keywords: Malware detection, computer security, information retrieval, static analysis

Mathematics Subject Classification 2010: 68-00, 68T30, 68U3

1 INTRODUCTION

Malware (or malicious software) is defined as computer software that has been explicitly designed to harm computers or networks. The number, power and variety of malware programs increases every year, as does their ability to overcome all kinds of security barriers[1].

Current commercial anti-malware solutions rely on a signature database [1]. A signature is a sequence of bytes that is always present within a malicious executable. To determine the signature for a new malicious executable and to counteract it, specialists must wait until the new malicious executable has damaged several computers or networks. After the signatures have been determined, suspicious files can be analyzed by comparing their bytes with the list of signatures. When a match is found, the file being tested will be identified as a malicious executable. This approach has proven to be effective when the threats are known beforehand.

However, malware writers use code obfuscation techniques [2] to hide the actual behaviour of their malicious creations [3, 4, 5, 6]. Examples of these obfuscation algorithms include *garbage insertion*, which consists of adding sequences that do not modify the behaviour of the program; *code reordering*, which changes the order of program instructions and *variable renaming*, which replaces one variable identifier with another [7].

The research community has proposed several approaches to deal with these obfuscation techniques. Sung et al. [8] and Xu et al. [9] introduced a method for computing the similarity between two executables by focusing on the degree of similarity within *syscall*[2] *sequences.* This approach offered limited performance because of its inability to maintain a low false positive ratio. Other approaches used the so-called *Control Flow Graph (CFG)* analysis. An example was introduced by Lo et al. [10] as part of the *Malicious Code Filter (MFC)* project. Their method sliced a program into blocks while looking for tell-tale signs (i.e., operations that change the state of a program such as network access events and file operations) in order to determine whether an executable was likely to be malicious or not. Bergeron et al. [11] presented several methods of disassembling binary executables, helped build a representation of the execution flow of binary executables and improved the slicing of a program into *idioms* (i.e., sequences of instructions). Christodorescu and Jha [12] proposed a method based on CFG to handle obfuscations in malicious software. Christodorescu et al. [13] improved on this work by including semantic-templates of malicious specifications. However, these approaches consume significant computational resources and the templates have to be constructed manually [14].

[1] Kaspersky Security Bulletin: Statistics 2008. Available online: `http://www.viruslist.com/en/analysis?pubid=204792052`

[2] A syscall or system call is the procedure through which an executable requests a service from the kernel of the operating system.

Given this background, a new system for the detection of malware variants is proposed that uses an architecture based on information retrieval. This architecture is composed of:

1. a malware database,
2. a framework for representing executables,
3. a component that transforms the executable to a query for the malware database and
4. a ranking function for the detection of malware variants.

The representation is based on *opcodes* (i.e., operational codes in machine language) and the frequency of appearance of opcode sequences. Bilar [15] statistically analyzed the ability of single opcodes to serve as the basis for malware detection and confirmed their high reliability in determining the maliciousness of executables. Later, a new study proved that detecting malware family variants was feasible using opcode sequences [16].

The following advances in the state of the art have been performed:

- An information-retrieval-based architecture for the development of a malware variant detection system and a description of how to adapt this information-retrieval-based approach for malware detection. How to use an opcode-sequence-frequency representation of executables to detect and classify malware variants is shown.
- A prototype of an information-retrieval-based malware detection system.
- An evaluation of the complete framework in terms of performance and malware variant detection capabilities.

The rest of this paper is organized as follows. Section 2 details the design of our information-retrieval-based malware detection system. Section 3 describes the implementation of the different components of our malware detection system. Section 4 describes the experiments performed and presents the results. Section 5 discusses the advantages and limitations of the proposed system. Finally, Section 6 concludes the paper and outlines avenues for future work.

2 METHOD DESCRIPTION

For the design of our information-retrieval-based malware detector, the formal definition of Information Retrieval (IR) proposed by Baeza-Yates and Ribeiro-Neto [17] is used. Adapting this definition to the malware detection context, an IR model is defined as a 4-tuple $[\mathcal{E}, \mathcal{Q}, \mathcal{F}, R(q_i, e_j)]$ where $\mathcal{E}$ is a set of representations of executables, in our case, representations of known malware variants; $\mathcal{Q}$ is a set of representations of user queries, in our case, representations of executable files under inspection; $\mathcal{F}$ is a framework for modelling executables, queries and their relationships and $R(q_i, e_j)$ is a ranking function that associates a real number with a query q_i, also called a *similarity function.*

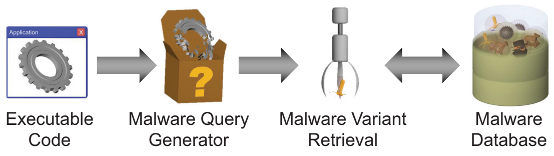


Fig. 1. General overview of the malware detector

Accordingly, our method is detailed in terms of these 4 components (see Figure 1). In particular, $\mathcal{F}$ details how our system models executables. $\mathcal{E}$ describes the *Malware Database* that contains the representations of known malware variants. $\mathcal{Q}$ determines how an executable in inspection is transformed into a query for the system by means of the *Malware Query Generator*. Finally, $R(q_i, e_j)$ details how the system finds the most similar executables to a query, in other words, how the systems detects malware through the *Malware Variant Retrieval* component.

The malware detection system starts with the *Malware Query Generator*. This component disassembles the executable being analyzed and generates a vectorial representation of it, posing as the query for the *Malware Database*. Next, through a similarity measure, the system retrieves the executables whose representations are selected by the *Malware Variant Retrieval* module. To this end, *similarity threshold* is defined so as to let the system select the malicious executables whose similarity ratio surpassed that threshold value. Finally, if no similar representation exists in the *Malware Database*, the executable under inspection will be declared as legitimate software. Otherwise, the executable is considered a variant of the malware family to which the most similar malicious executables belong.

2.1 Framework for Modeling Executables

This component determines how binaries are modelled in the proposed information-retrieval-based malware detector. In particular, how to mine the relevance of each opcode to avoid any distortion produced by the most common opcodes, such as `mov` or `push`, and how to represent executables through opcode sequences is described.

Bilar [15] investigated the ability of operational codes to detect malware. The study concluded that opcodes are able to reveal significant statistical differences between malware and legitimate software and that rare opcodes are better predictors than common opcodes.

This previous study is extended by providing a method that measures the relevance of individual opcodes. To this end, a methodology is employed that computes a weight for each operational code. This weighting represents the relevance of the opcode to discriminate between malicious and benign executables based on the frequency of occurrence of opcodes in malicious and benign executables. Malware from the VxHeavens website[3] was collected to assemble a malware dataset of 13 189 malware executables. This dataset contained only *Portable Executable* (PE)[4] files, and it was made up of different types of malicious software (e.g., computer viruses, Trojan horses and spyware). For the benign software dataset, 13 000 executables from our computers were collected. This benign dataset included text processors, drawing tools, Windows games, Internet browsers and PDF viewers. A confirmation of whether the benign executables were not infected has been performed because any infections would have distorted the results. An analysis of the benign files using *Eset Antivirus*[5] was performed.

The method for computing the relevance of opcodes is composed of the following steps:

Disassembling the executables: NewBasic Assembler[6] was used as the main tool for obtaining the assembly files.

Generation of opcode profile file: Using the generated assembly files, opcode profiles were built. Each file contains a list with the operational code and the times that each opcode appears within both the benign software dataset and the malicious software dataset.

Computation of opcode relevance: The relevance of each opcode was computed based on the frequency with which it appears in each dataset. *Mutual Information* [18] was used (shown in Equation (1)) to measure the statistical dependence of the two variables:

$$I(X;Y) = \sum_{y\epsilon Y} \sum_{x\epsilon X} p(x,y) \log\left(\frac{p(x,y)}{p(x)\cdot p(y)}\right) \tag{1}$$

where X is the opcode frequency and Y is the class of the file (i.e., malware or benign software), $p(x,y)$ is the joint probability distribution function of X and Y, and $p(x)$ and $p(y)$ are the marginal probability distribution functions of X and Y. Note that this weight only measures the relevance of a single opcode and not the relevance of an opcode sequence.

Once the mutual information between each opcode and the executable class had been computed, an opcode relevance file is created by sorting the opcodes by relevance. The opcode frequency file was saved so that further calculations of the relevance of the opcodes may be calculated using other measures such as the gain ratio [19] or chi-square [20].

This list of opcode relevance helped with more accurate malware detection because it is possible to weight the final representation of executables using the cal-

[3] `http://vx.netlux.org/`

[4] The Portable Executable (PE) format is a file format for executables, object code and DLLs, used in Microsoft Windows operating systems.

[5] http://www.eset.com/

[6] `http://www.frontiernet.net/~fys/newbasic.htm`

culated opcode relevances and reducing noise from irrelevant opcodes. Specifically, the most common opcodes, such as `push`, `mov` or `add`, tended to be weighted low in the results.

These weights may be considered as a replacement for the Inverse Document Frequency (IDF) measure [21] used in the Vector Space Model [22, 23] for information retrieval. IDF weighs terms within documents based on the frequency of occurrence in the whole document set. Our method performs a similar task by using mutual information instead of simply counting occurrences. It is also easier to update, using the total number of opcode occurrences in each dataset (both malicious and benign).

To represent executables using opcodes, the *opcode-sequences* and their frequency of appearance were extracted. More specifically, a program ρ is a set of ordered opcodes o, $\rho = (o_1, o_2, o_3, o_4, \ldots, o_{\ell-1}, o_\ell)$, where ℓ is the number of instructions I of the program ρ. An opcode sequence s is defined as a subgroup of opcodes within the executable file where $s \subseteq \rho$; it is made up of opcodes o, $s = (o_1, o_2, o_3, \ldots, o_{m-1}, o_m)$ where m is the length of the sequence of opcodes s.

Although Bilar [15] only proved that single opcodes were good predictors for malware detection, this approach is generalized and the use of sequences of opcodes is proposed instead. The main assumption is that longer opcode sequences provide a lower false positive ratio (i.e., benign executables misclassified as malware) whereas shorter opcode sequences identify more malware variants. Therefore, by using opcode sequences instead of single opcodes, it is possible to configure the behaviour of the detector. However, it is difficult to establish an optimal value for the lengths of the sequences: a small value will fail to detect complex malicious blocks of operations whereas long sequences can easily be avoided with simple obfuscation techniques. Besides, long opcode sequences will introduce a high performance overhead.

Afterwards, the frequency of occurrence of each opcode sequence within the file is computed by using *term frequency* (tf) [22] (shown in Equation (2)) that is a weight widely used in information retrieval [24]:

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \tag{2}$$

where i is the index for the opcode sequence and j is the index for the executable. $n_{i,j}$ is the number of times the sequence $s_{i,j}$ (the i^{th} sequence within the executable j) appears in an executable j, and $\sum_k n_{k,j}$ is the total number of sequences in the executable j (in our case the total number of possible opcode sequences).

This measure is computed for every possible opcode sequence of fixed length n, acquiring a vector $\vec{v}$ of the frequencies of opcode sequences $s_{i,j} = (o_1, o_2, o_3, \ldots, o_{n-1}, o_n)$. The frequency of occurrence of this opcode sequence is weighted using the relevance weights.

Weighted Term Frequency (WTF) is defined as the result of weighting the relevance of each opcode when calculating the term frequency. Specifically, the WTF is computed as the product of sequence frequency and the calculated weight of every opcode in the sequence:

$$wtf_{i,j} = tf_{i,j} \cdot \prod_{o_z \epsilon S} \frac{weight(o_z)}{100} \tag{3}$$

where $weight(o_z)$ is the calculated weight, by means of mutual information gain, for the opcode o_z and $tf_{i,j}$ is the *sequence frequency measure* for the given opcode sequence.

Applying the calculated weighted sequence frequencies, a vector $\vec{v}$ is obtained, composed of weighted opcode-sequence frequencies, $\vec{v} = ((os_1, wtf_1), (os_2, wtf_2), \ldots,$ $(os_{n-1}, wtf_{n-1}), (os_n, wtf_n))$, where os_i is the opcode sequence and wtf_i is the weighted term frequency for that particular opcode sequence. Using the resultant vector representation of the files, it is possible to compute the similarity between two input files.

2.2 Malware Database

This component stores the representation of different variants from different malware families, as a *corpus* of executables $\mathcal{E}$ to retrieve the most similar variants to an executable under inspection. To this end, our approach stores the executables and their opcode sequences. Because the system uses these representations to detect malware variants, they can be considered as *signatures*.

The data model [25] of this database describes how the data is represented in our system. Figure 2 shows the proposed data model.

The entity *Opcode* represents the unique opcodes of machine language. *Opcode_id* is the identification for the opcode. The attribute *Mnemonic* is the assembly representation of the machine operation, whereas *Weight* is the calculated relevance for that particular opcode.

The entity *Sequence model* stores the relationships between the different opcodes and the sequences they form. *Sequence_id* is the identification for the opcode sequence, *Opcode_id* is the identification of the opcode and *Pos* is the position of the opcode within the sequence.

The entity *Opcode sequence* represents the different opcode sequences. *Sequence_id* is the identification for the unique opcode sequence and *Length* establishes the opcode sequence length.

Executable model is an entity that stores the number of times the sequences within an executable appear. *Executable_id* is the identification of the executable. *Sequence_id* is the identification of the opcode sequence and *Sequence frequency* is the number of times the particular sequence (identified by *Sequence_id*) appears within the executable (identified by the*Executable_id*).

The entity *Malicious executable* stores the information of the executables that are malware. *Name* establishes the particular labelling of that executable and *Family_id* is the identification of the malware family the executable belongs to.

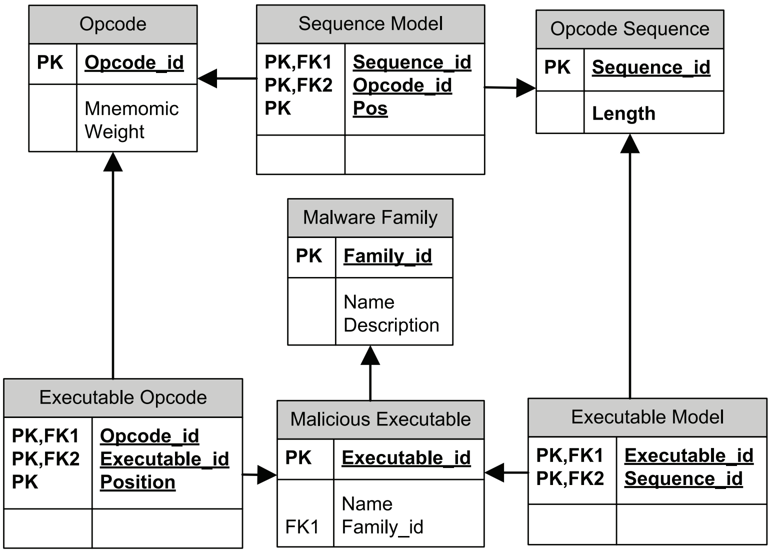


Fig. 2. Proposed data model for the malware database. PK stands for primary key whereas FK means foreign key.

Malware family stores the different malware families. *Name* is the name of the family and *Description* is a brief abstract of the functionality of the family.

Finally, *Executable opcode* represent the opcodes an executable is composed of *Executable_id* is the identification of the executable, *Opcode_id* is the identification of the opcode and *Pos* stores the position of the opcode within the executable.

2.3 Malware Query Generator

This component generates a query in an *opcode sequence* format of the binary executable under inspection. The resultant query is the key element for the variant malware retrieval system.

Since the original source code is not available for the executables, the first step is to disassemble them. To this end, *NewBasic Assembler* was employed. However, some executables are packed and it would not be able to extract the real assembly code. In those cases, it would need a previous unpacking step to the proposed framework.

Once the source code is available, the opcode sequence frequency representation of the executable is extracted. In this step, it is required to configure the length of the sequence or even utilize different lengths and combine them into a unique representation of the executable, called *recursive representation*. For instance, a recursive representation of length 3 includes the opcode sequences of length 1, 2 and 3.

Finally, using the extracted frequencies, they are weighted with the opcode relevances forming the WTF representation. This vector is considered the query for the malware database.

2.4 Malware Variant Retrieval

The retrieval of the most similar malware variants to an executable under inspection is the detection step of our system. Once the query vector composed of weighted opcode sequence frequencies is generated, it is compared with the malicious executables stored in the malware database.

In this way, it is possible to measure how similar the query is to the executables within the malware database. Representing the two files as two vectors $\vec{v}$ and $\vec{u}$, it is possible to measure their similarity by means of the cosine similarity [26]:

$$sim(\vec{v},\vec{u}) = \cos(\theta) = \frac{\vec{v}\cdot\vec{u}}{||\vec{v}||\cdot||\vec{u}||} \tag{4}$$

where $\vec{v}\cdot\vec{u}$ is the inner product of $\vec{v}$ and $\vec{u}$ whereas $||\vec{v}||\cdot||\vec{u}||$ is the cross product of $\vec{v}$ and $\vec{u}$.

This value ranges from 0 to 1, where 0 means that the two executables are completely different (i.e., the vectors are orthogonal between them) and 1 means that the executables are equivalent. To retrieve the most similar executables within the malware database, a similarity threshold above which the system considers an executable as a malware variant must be determined first.

3 IMPLEMENTATION DETAILS

In this section, the implementation of the malware scanner prototype, called N-Opcode Analyzer (NOA), is described based on the variant analysis system described in Section 2.

An end-host application using the Microsoft .NET platform (Microsoft .NET Framework 3.5 and SQL Server 2008) has been developed, that allows the user to scan directories recursively and find malware variants present in the system.

3.1 Representation Framework

The representation framework provides functionality to generate malware representations, as shown in Section 2.1, both for database population and query composition. This process consists of three steps.

Disassembly: The first step relies on an external disassembler and a module that processes the output assembly code from the disassembler to generate the malware representation. The freeware disassembler *NewBasic Assembler* was employed, by launching the program from our application and setting it up to write the results over a temporary text file that is read afterwards.

Generation of the *Term Frequency* (TF): In this step, a list containing the frequency for each possible opcode sequence is created. Note that sequences can be of length n or $1..n$ if recursive option is selected. To calculate this frequency value, sequences are generated from the previously disassembled file, counting the number of occurrences in the full program. After that, it is necessary to divide each counter by the total number of different sequences in the file.

Generation of the *Weighted Term Frequency* (WTF): Finally, each sequence frequency is updated according to opcode relevance weight values. For this, the frequency is multiplied by the weight value of each opcode in the sequence. These weight values are previously calculated and stored in the database.

3.2 Data Model

To improve system efficiency, the database model has been modified. First, the storage of the full list of opcodes associated to each malware variant saved on the database is avoided. Instead, the representations for a sequence length from 1 to 5 are calculated and saved. Longer sequences introduce a considerable overload on the system. Given that analyses are not going to be performed for sequences longer than 5 opcodes, full storage for further calculations is unnecessary.

The table *MaliciousExecutable* contains the list of malware executables stored in the database, consisting of an incremental numeric key and a name to designate the variant. *ExecutableModel* stores the malware representations, associating a weighted frequency value to each sequence and executable. Note that the system does not make use of any relation between sequences and opcodes. This approach avoids constraint checking performed by the database management system. Alternatively, the system generates a key for each sequence based on the keys of the opcodes that compose it. Assuming a maximum sequence length of 5 and a maximum number of weighted opcodes lower than 999, it can be stated that the maximum value for this key is 15 digits long, which can be contained in an integer field of 64 bits. Opcodes with a weight value of 0 will cause a WTF of 0 for the sequences they are part of. Subsequently, these sequences will not be stored in the database and longer keys will not be necessary. The sequence key is calculated applying: $SequenceKey = \sum_{i=1}^{n} OpcodeKey_i \cdot 1\,000^{i-1}$, where n is the chosen opcode sequence length and $OpcodeKey_i$ is the i^{th} opcode key in the sequence.

TempAnalysis is used to store temporary representations of malware executables for further comparison with the representations stored in the database. Keys for this table are equally generated based on opcode keys.

The table *Opcode* holds weight values, as well as the mnemonic for each opcode key.

Finally, *Settings* is used to store several parameters configurable by the end-user, which will be applied during the analysis process.

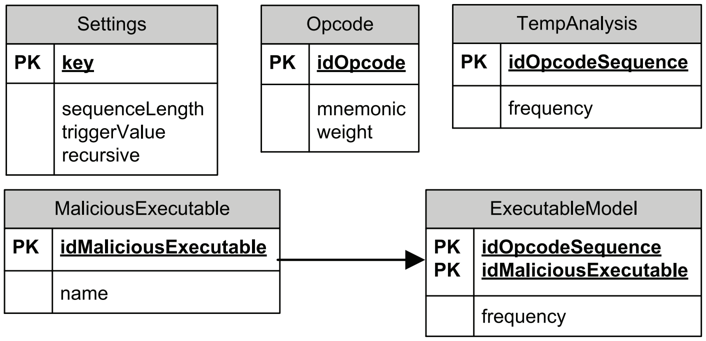


Fig. 3. Database diagram finally utilized in the prototype implementation. This version is simplified with respect to the data model proposed in Section 2 due to efficiency requirements.

3.3 Database Query

Once a representation is generated by the representation framework, it is stored in the table *TempAnalysis*.

```
CREATE PROCEDURE
[dbo].[analyzeExecutable]
   @triggerValue REAL,
   @sequenceLength int,
   @recursive bit
)
AS
BEGIN
   SELECT temp.name,
temp.similarity FROM
   (SELECT name,
dbo.cosineSimilarity
   (idMaliciousExecutable,
@sequenceLength, @recursive)
AS similarity                  FROM
dbo.maliciousexecutable) temp
   WHERE temp.similarity >=
@thresholdValue
   ORDER BY temp.similarity desc;
END
```

Fig. 4. Stored procedure used to retrieve a list of previously stored similar malware variants, implemented in Transact-SQL

```
CREATE FUNCTION
[dbo].[cosineSimilarity]
(
   @idExecutable int,
   @sequenceLength int,
   @recursive bit
)
RETURNS Real
AS
BEGIN
   DECLARE @Result Real
   SET @Result =
dbo.dotProduct(@idExecutable) /
   dbo.moduleProduct(@idExecutable,
   @sequenceLength, @recursive);
   RETURN @Result
END
```

Fig. 5. Cosine similarity function implemented in Transact-SQL

Next, the stored procedure shown in Figure 4 is executed, calculating the cosine similarity among the sequences stored on *TempAnalysis* and the ones stored in the table *ExecutableModel*. This method returns a list of possible matches in descending order of similarity. The stored procedure combines rows in both tables joining them by their key value and filtering *ExecutableModel* by each executable key. Thereby, it computes the cosine similarity for each executable stored in the database.

To calculate cosine similarity, this stored procedure employs the function *cosineSimilarity* (shown in Figure 5), which calls to *dotProduct* (Figure 6) and *moduleProduct* (Figure 7) functions. The *dotProduct* function sums up the values resulting from multiplying the opcode sequence frequencies in each table. The *moduleProduct* function returns the product of the modules of each representation vector. In the case of *ExecutableModel*, it is necessary to filter the table by *idMaliciousExecutable* and sequence numbers. To this end, the system calculates the minimum and maximum possible *IdOpcodeSequence* values for the selected values of sequence length and recursion parameter. If any match has a similarity rate higher than the previously specified threshold similarity value, the sample will be considered as a malware instance, member of the family of the executable it matches with. In case there are various matches with a similarity rate higher than the threshold value, the system will select the family of the most similar variant.

3.4 Graphical User Interface

The Graphical User Interface (GUI) designed for this prototype allows the end-user to make use of the described analysis system. This GUI intends to be a complete anti-malware system. However, since it is highly experimental, it lacks

```
CREATE FUNCTION [dbo].[dotProduct]
(
   @idExecutable int
)
RETURNS REAL
AS
BEGIN
   DECLARE @Result REAL;
   SELECT @Result =
SUM(a.[frequency] *b.[frequency])
   FROM dbo.tempAnalysis AS a
INNER JOIN
   dbo.executablemodel AS b
   ON a.[idOpcodeSequence] =
b.[idOpcodeSequence]
   WHERE b.[idMaliciousExecutable]
= @idExecutable;
   RETURN @Result;
END
```

Fig. 6. Dot product function implemented in Transact-SQL

some classic functionality such as automatic scan of external drives or quarantine folder.

Figure 8 shows the implemented GUI. The first tab-page (see Figure 8 a)) is used to scan directories recursively. When the user selects a directory on the computer, the analysis process will start. A list at the bottom of the window shows the found variants, indicating the malware family they pertain to.

The second tab-page (refer to Figure 8 b)) permits the user to update the database with new representations of malware families. Similarly, it is necessary to select a directory. All executables inside the folder will be represented and added to the database as malware executables.

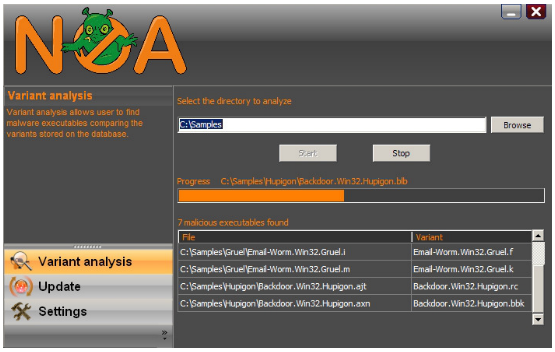
At last, the third tab-page (shown in Figure 8 c)) is used for setting some parameters regarding the analysis process: sequence length, recursion and threshold value for representation matching.

4 EMPIRICAL VALIDATION

In this section, the process conducted to validate our malware variant detection prototype (called *N-Opcode Analyzer*: NOA) is described. This experiment shows that NOA can detect malware variants within a common computer environment.

```
CREATE FUNCTION [dbo].[moduleProduct]
(
   @idExecutable int,
   @sequenceLength int,
   @recursive bit
) RETURNS Real
AS
BEGIN
   DECLARE @Result REAL;
   DECLARE @a REAL;
   DECLARE @b REAL;
   DECLARE @idMinValue bigint;
   DECLARE @idMaxValue bigint;
   SET @idMaxValue =
(POWER(CAST (1 000 CAST bigint),
CAST(@sequenceLength CAST bigint)))
-1;
   IF @recursive = 0
      SET @idMinValue =
POWER(CAST(1 000 AS bigint),
CAST(@sequenceLength AS bigint) -
1);
   ELSE
      SET @idMinValue = 0;
   SELECT @a = SQRT(SUM(a.[frequency]
*a.[frequency]))
   FROM [NOA].[dbo].[tempAnalysis] AS
a;
   SELECT @b = SQRT(SUM(b.[frequency]
*b.[frequency]))
   FROM [NOA].[dbo].[executablemodel]
AS b
   WHERE b.[idMaliciousExecutable] =
@idExecutable AND
   b.[idOpcodeSequence] <=
@idMaxValue AND b.[idOpcodeSequence]
>= @idMinValue;
   SET @Result = @a * @b;
   RETURN @Result;
END
```

Fig. 7. Module product function implemented in Transact-SQL



Variant analysis

Variant analysis allows user to find malware executables comparing the variants stored on the database.

Select the directory to analyze

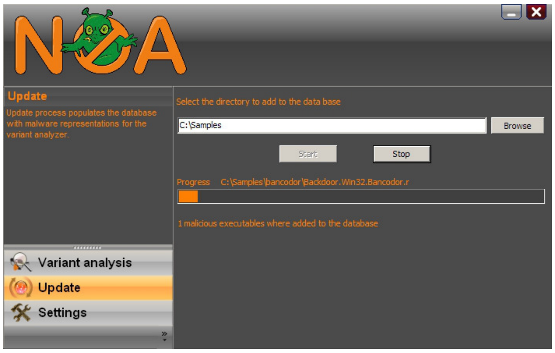
C:\Samples

Progress C:\Samples\Hupigon\Backdoor.Win32.Hupigon.blb

7 malicious executables found

File	Variant
C:\Samples\Gruel\Email-Worm.Win32.Gruel.i	Email-Worm.Win32.Gruel.f
C:\Samples\Gruel\Email-Worm.Win32.Gruel.m	Email-Worm.Win32.Gruel.k
C:\Samples\Hupigon\Backdoor.Win32.Hupigon.ajt	Backdoor.Win32.Hupigon.rc
C:\Samples\Hupigon\Backdoor.Win32.Hupigon.axn	Backdoor.Win32.Hupigon.bbk

a)



Update

Update process populates the database with malware representations for the variant analyzer.

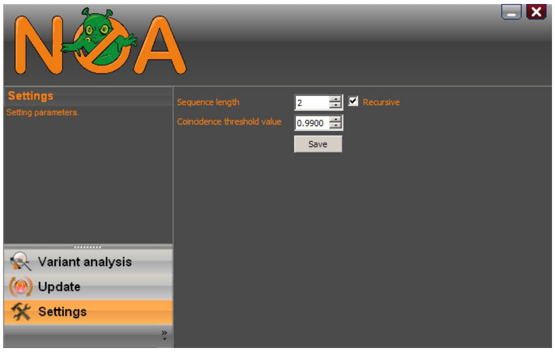
Select the directory to add to the data base

C:\Samples

Progress C:\Samples\bancodor\Backdoor.Win32.Bancodor.r

1 malicious executables where added to the database

b)



Settings

Setting parameters.

Sequence length 2 [✓ Recursive]

Coincidence threshold value 0.9900

c)

Fig. 8. The Graphical User Interface of the implemented prototype. a) Variant analysis tab page. This tab is used to scan recursively a directory to find if any malware variant is present in that directory. b) Update database tab page. This tab is used to populate the database with malware variants. c) Settings tab. This tab is used to set the parameters regarding the representation framework and the confidence threshold to determine malware variants.

4.1 Experiment Design

To perform the experiment, a malware dataset composed of a set of malware variants from 5 well-known malware families: Antilam, Bancodor, Gruel, Hupigon and PcClient (for further details regarding the malware dataset, refer to Tables 1, 2, 3, 4, 5) was used. These samples were downloaded from VxHeavens. As the benign software dataset, a set of 1 000 benign executables was selected from a clean Microsoft Windows XP installation (for more details for the legitimate dataset refer to Table 6).

Samples in database	Samples to be analyzed
Backdoor.Win32.Antilam.13.b	Backdoor.Win32.Antilam.20.q
Backdoor.Win32.Antilam.14.i	
Backdoor.Win32.Antilam.20.n	
Backdoor.Win32.Antilam.20.p	
Backdoor.Win32.Antilam.20.s	
Backdoor.Win32.Antilam.gen	
6	1

Table 1. Antilam family samples used for the described experiment

Samples in database	Samples to be analyzed
Backdoor.Win32.Bancodor.b	Backdoor.Win32.Bancodor.c
Backdoor.Win32.Bancodor.d	Backdoor.Win32.Bancodor.r
Backdoor.Win32.Bancodor.e	
Backdoor.Win32.Bancodor.f	
Backdoor.Win32.Bancodor.j	
Backdoor.Win32.Bancodor.m	
Backdoor.Win32.Bancodor.t	
Backdoor.Win32.Bancodor.u	
Backdoor.Win32.Bancodor.x	
9	2

Table 2. Bancodor family samples used for the described experiment

First, the PEiD[7] tool was used to check that the malware executables were not packed since our static representation would not be able to deal with them. As it was done when computing the relevance of opcodes, an analysis of the benign files was performed using Eset Antivirus to check that the benign executables were not infected and, therefore, validate the legitimacy of the benign dataset.

Hereafter, a subset of malicious samples consisting of 80 % of the malicious executables from each malware family is selected to populate the database. The

[7] `http://www.peid.info/`

Samples in database	Samples to be analyzed
Email-Worm.Win32.Gruel.a	Email-Worm.Win32.Gruel.c
Email-Worm.Win32.Gruel.b	Email-Worm.Win32.Gruel.i
Email-Worm.Win32.Gruel.e	Email-Worm.Win32.Gruel.m
Email-Worm.Win32.Gruel.f	
Email-Worm.Win32.Gruel.g	
Email-Worm.Win32.Gruel.h	
Email-Worm.Win32.Gruel.j	
Email-Worm.Win32.Gruel.k	
Email-Worm.Win32.Gruel.l	
Email-Worm.Win32.Gruel.n	
10	3

Table 3. Gruel family samples used for the described experiment

remaining 20 % of the malicious samples of each malware family is used to analyze them and, thus, validate NOA. Besides, the 1 000 benign executables were used to check the number of false positives (legitimate applications misclassified as malware) raised by NOA.

In our validation, the representations with an opcode-sequence length from 1 to 3 combining *Recursive* (R) and *Not Recursive* (NR) approaches were used. A recursive representation of $n = 3$ means that sequences with a length of 1, 2 and 3 are used. Since our main goal is to provide a fast malware detector, longer opcode sequences were not used because of the performance overhead they introduce. In fact, several preliminary tests were performed with opcode sequences of length 4 and 5 but the experimental machine could not cope with the work in a reasonable time.

To conduct the validation, a simple routine in Microsoft .NET Framework 3.5 was implemented that calls the analysis process by using the proposed architecture framework. In this way, a complete analysis process over the whole dataset of executables is repeated applying different values to each parameter. In the first experiment, the parameter ranges shown in Figure 7 were used. In this case, the results obtained show that threshold values under 0.99 were not precise enough for executable comparison, producing good detection rates but, unfortunately, also a high false positive ratio. Therefore, a second experiment was performed with different threshold values (see Figure 8) to determine more accurate parameter values.

As the experimental environment, two different machines were used: the machine where the database was installed and the machine where NOA was going to run. As the database server, an Intel Core i7 940 clocked at 2.93 GHz and 8 GB of RAM memory was used. More accurately, the database server was a SQL Server 2008 SP1. As the analysis platform, a VMWare[8] virtual machine hosted in an Intel Core i7 940 clocked at 3.07 GHz and 12 GB of RAM memory was used. The virtual machine

[8] `www.vmware.com`

Samples in database	Samples to be analyzed
Backdoor.Win32.Hupigon.abc	Backdoor.Win32.Hupigon.ajt
Backdoor.Win32.Hupigon.abu	Backdoor.Win32.Hupigon.axn
Backdoor.Win32.Hupigon.aj	Backdoor.Win32.Hupigon.blb
Backdoor.Win32.Hupigon.akx	Backdoor.Win32.Hupigon.dl
Backdoor.Win32.Hupigon.amc	Backdoor.Win32.Hupigon.gb
Backdoor.Win32.Hupigon.are	Backdoor.Win32.Hupigon.up
Backdoor.Win32.Hupigon.awl	Backdoor.Win32.Hupigon.vw
Backdoor.Win32.Hupigon.awn	
Backdoor.Win32.Hupigon.bbk	
Backdoor.Win32.Hupigon.blh	
Backdoor.Win32.Hupigon.bqz	
Backdoor.Win32.Hupigon.bto	
Backdoor.Win32.Hupigon.ccf	
Backdoor.Win32.Hupigon.dh	
Backdoor.Win32.Hupigon.eg	
Backdoor.Win32.Hupigon.ej	
Backdoor.Win32.Hupigon.fo	
Backdoor.Win32.Hupigon.g	
Backdoor.Win32.Hupigon.j	
Backdoor.Win32.Hupigon.lp	
Backdoor.Win32.Hupigon.qk	
Backdoor.Win32.Hupigon.r	
Backdoor.Win32.Hupigon.rc	
Backdoor.Win32.Hupigon.ri	
Backdoor.Win32.Hupigon.sc	
Backdoor.Win32.Hupigon.ub	
Backdoor.Win32.Hupigon.vh	
Backdoor.Win32.Hupigon.xr	
Backdoor.Win32.Hupigon.ze	
29	7

Table 4. Hupigon family samples used for the described experiment

specification is as follows: 1 processor, 1 GB of RAM memory and Windows XP SP3 as operative system. In particular, to evaluate the performance and precision of NOA, the following aspects were measured:

Time results: The processing overhead of NOA in 4 different aspects was measured:

Total analysis time: The total time required to analyze an executable. It includes disassembling, representation and comparison time.

Disassembling time: The percentage of the total time employed to disassemble the executable under inspection. Note that this time is not de-

Samples in database	Samples to be analyzed
Backdoor.Win32.PcClient.ah	Backdoor.Win32.PcClient.bn
Backdoor.Win32.PcClient.au	Backdoor.Win32.PcClient.du
Backdoor.Win32.PcClient.b	Backdoor.Win32.PcClient.hw
Backdoor.Win32.PcClient.bg	Backdoor.Win32.PcClient.pj
Backdoor.Win32.PcClient.bm	
Backdoor.Win32.PcClient.dq	
Backdoor.Win32.PcClient.fb	
Backdoor.Win32.PcClient.h	
Backdoor.Win32.PcClient.ik	
Backdoor.Win32.PcClient.jl	
Backdoor.Win32.PcClient.m	
Backdoor.Win32.PcClient.nu	
Backdoor.Win32.PcClient.qe	
Backdoor.Win32.PcClient.sd	
Backdoor.Win32.PcClient.wj	
Backdoor.Win32.PcClient.xc	
16	4

Table 5. PcClient family samples used for the described experiment

Samples in database	Samples to be analyzed
0	1 000

Table 6. Legitimate samples used for the described experiment

pendent on our solution because the NewBasic Assembler was used to this end.

Representation time: The percentage of the total time utilized to represent an executable, transforming it to the proposed representation.

Comparison time: The percentage of the total time used to compare the executable under inspection with representations already stored in the database. This step also ranks the possible matches and selects the most certain one.

Accuracy results: The accuracy results were evaluated by measuring False Negative Ratio (FNR) and False Positive Ratio (FPR). FNR and FPR measures are

Parameters	**Possible Values**
Sequence Length	1, 2, 3
Recursion	Recursive (R), Not Recursive (NR)
Threshold	0.99, 0.98, 0.97, 0.96, 0.95

Table 7. Parameters used in the first experiment. All the possible values are permuted in 30 analysis processes

Parameters	**Possible Values**
Sequence Length	1, 2, 3
Recursion	Recursive (R), Not Recursive (NR)
Threshold	0.999, 0.998, 0.997, 0.996, 0.995, 0.994, 0.993, 0.992, 0.991

Table 8. Parameters used in the second experiment. All the possible values are permuted in 54 analysis processes

a common standard to evaluate the performance of malware detection because they are the two types of errors a malware detector can produce. All the results refer to the ability of the proposed method not to only detect malware but to detect the malware family of analyzed variant. In particular, FNR is defined as:

$$FNR(\beta) = \frac{FN}{FN + TP} \tag{5}$$

where TP is the number of malware cases correctly classified (true positives) and FN is the number of malware cases misclassified as legitimate software (false negatives).

On the other hand, FPR is defined as:

$$FPR(\alpha) = \frac{FP}{FP + TN} \tag{6}$$

where FP is the number of benign executables incorrectly detected as malware while TN is the number of legitimate executables correctly classified.

Both measures establish the cost of misclassification. Therefore, it is important to set the cost of false negatives and false positives; in other words, to establish whether it is better to classify a malware as legitimate or to classify a benign software as malware. In particular, since our framework is devoted to detect malware variants, one may think that it is more important to detect more malware than to minimize false positives. However, commercial products tend to consider achieving a low false positive ratio more important: users can be bothered if their legitimate applications are constantly flagged as malware. Hence, it can be considered that the importance of the cost must be established depending on the purposes NOA will be used with. If it is used as a complement to standard anti-malware systems then the system should focus on minimizing false positives. Otherwise, if the framework is used by antivirus laboratories to decide which executables should be further analyzed then the system should minimize false negatives (or, in other words, maximize the detection of malware).

To this end, the system can apply (i) whitelisting and (ii) blacklisting. White and black lists store a signature of an executable in order to be flagged either as malware (blacklisting) or benign software (whitelisting) without a further analysis.

4.2 Results

In this section, the obtained results are shown. In this way, Figure 9 shows the times/opcode sequence length ratio to represent the file analysis overhead, Figure 10 shows the proportion of the three different phases of the total analysis (disassembling, representation and comparison), Figure 11 shows the relation between file size and processing time and Figure 12 shows both FNR and FPR for each parameter set used in the experiment.

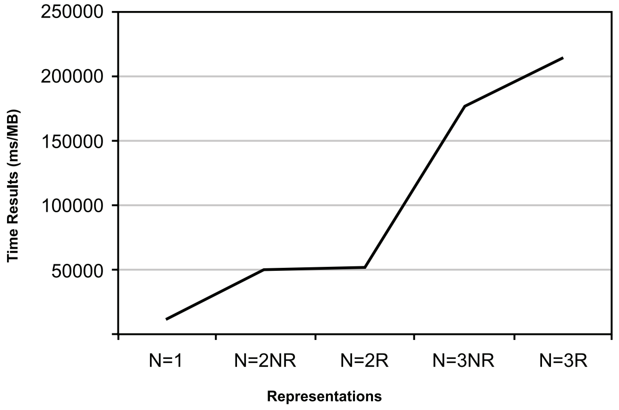


Fig. 9. Time results for each sequence length value, expressed in milliseconds, employed to analyze a megabyte of executable data. It can be noticed that sequence length exponentially increases required processing times. NR stands for *not recursive* representation whereas R stands for a *recursive* representation.

Total analysis time depends on 3 different parameters:

1. disassembly time, which is given by the disassembler tool used by our prototype;
2. representation time, which depends on the size of the sequence length (the higher the length, the higher the number of possible sequences in the representation) and
3. comparison time, which depends on the size of the vector generated in the representation phase and, thus, on the sequence length.

Comparison time also depends on the number of malware samples saved on the database. Note that this last step is completely performed by the database management system (in our case, SQL Server 2008).

It can be noticed that total time increases with the size of the executable (refer to Figure 11). Comparison time takes a greater portion of the analysis time (refer to Figure 10).

Likewise, the total time increases exponentially with sequence length, as shown in Figure 9. Although the complexity of our prototype is highly dependent on these

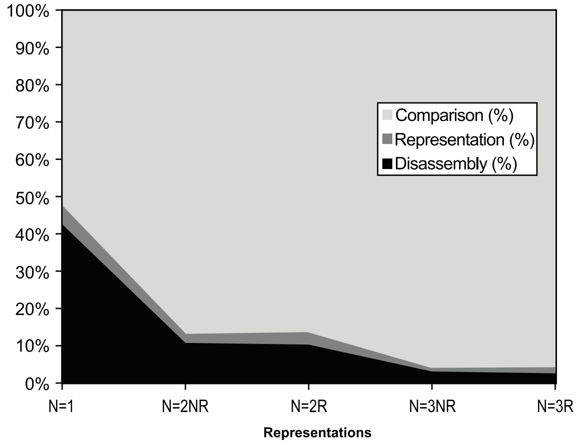


Fig. 10. The proportion of the analysis time dedicated to each phase. The time dedicated for the comparison step increases exponentially along with n whereas required disassembly and representation proportion decreases. NR stands for *Not Recursive* representation whereas R stands for *Recursive* representation.

three parameters, the analysis is performed relatively quickly. In particular, the average time of the analysis per file, without taking into account its size, was 563.37 ms for $n = 1$, 2 106.94 ms for $n = 2$, 2 207.44 ms for $n = 2$ recursive, 7 554.83 ms for $n = 3$ and 8 930.07 ms for $n = 3$ recursive.

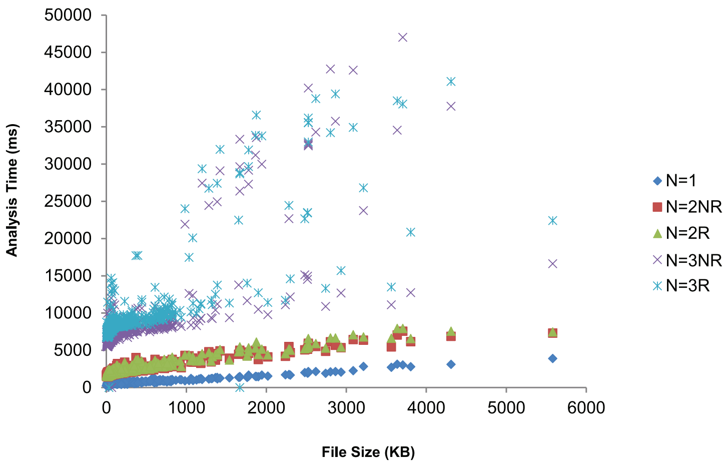


Fig. 11. Relation between file size and required time for the analysis. NR stands for *Not Recursive* representation whereas R stands for *Recursive* representation.

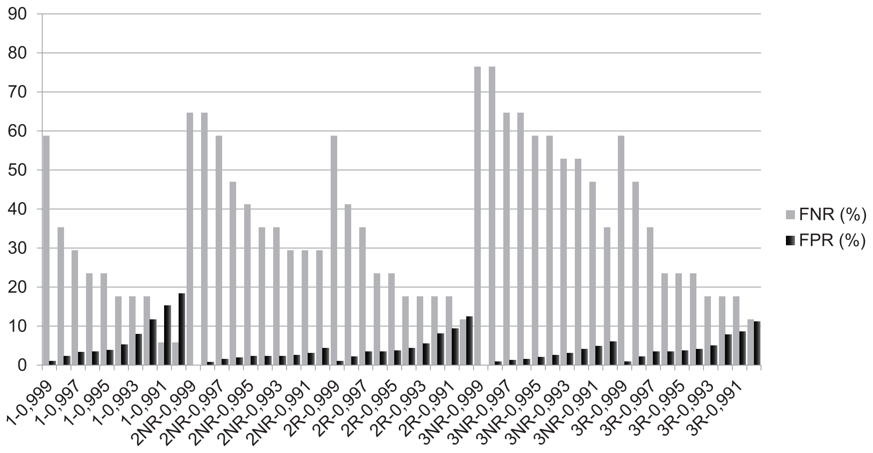


Fig. 12. FPR and FNR results. *NR* stands for *Not Recursive* representation whereas *R* stands for *Recursive* representation.

Regarding precision results, it is straightforward that higher threshold values will incur in a lower FPR and higher FNR (executables must be more similar to be identified) whereas lower thresholds cause the opposite effect. Consequently, it is mandatory to select a proper threshold value to equilibrate both rates. At the contrary, the most appropriate sequence length and recursion parameter values also have to be selected. Detection rates do not vary perceptively for sequence lengths higher than 2 while detection time increases significantly (see Figure 9). Specifically, in order to guarantee a 0.0 % of FPR, to generate the representation of the executables in their not recursive configuration is needed, with $n = 2$ or $n = 3$ and a confidence threshold of 0.999. However, this representations raised a high FNR: 64.7 % for $n = 2$ and 76.47 % for $n = 3$. On the other hand, the lowest FNR was achieved by the configuration $n = 1$ not recursive and 0.991 as threshold: 5.88 %. This configuration did not raise a very high amount of false positives: only the 15.2 % of the legitimate executables were misclassified as malware. Another interesting configuration is recursive $n = 2$ with a confidence threshold of 0.994: 17.64 of FNR and 4.3 % of FPR.

As conclusion, in our evaluation of NOA, a higher value of n always implies higher FNRs and a higher processing overhead on the analysis. However, it does not imply, at least not strongly, lower FPRs. Besides, the obtained accuracy does not show any improvement and, therefore, the system does not benefit of using longer opcode sequences. Overall, our solution has demonstrated that, with the proper configuration, NOA can be used to detect variants of malware and still be fast enough to be used in an anti-malware solution.

5 DISCUSSION

Malware detection can be categorized into static detection and dynamic detection. Dynamic detection analyzes the behaviour of an executable monitoring its execution in a contained environment whereas static detection analyzes it without executing it. Our work falls in the category of static analysis of malware.

In this category, the work of Christodorescu and Jha [12] and its improved semantic version by Christodorescu et al. [13] is very significant in this area. Their method was able to provide a higher level of representation than signatures and it was specific of a malicious behaviour. In this way, their method was able to identify, for instance, whether an executable connects to the Internet or has mail bomber capacity. However, there were several limitations. First of all, the approach was very time consuming and the representation templates require a manual step. Besides, the system identifies only behaviour and not malware families, complicating the disinfection step.

Other approaches have been proposed that use byte n-grams to detect unknown malware. Schultz et al. [27] were the first to introduce the concept of applying machine-learning algorithms for the detection of malware based on their respective binary codes. They applied different classifiers to three types of feature sets:

1. program headers,
2. strings and
3. byte sequences.

Later, Kolter et al. [28] improved Schulz's results [27], by applying n-grams (i.e., overlapping byte sequences) instead of non-overlapping sequences. This approach employed several algorithms, achieving the best results with a boosted[9] decision tree. Substantial research has focused on n-gram distributions of byte sequences and machine-learning algorithms [30, 31, 32]. However, the representativeness of bytes by themselves renders these approaches easily surpassed by only changing the compiler.

Since opcode sequences can be considered somehow as structural information, PE-Miner [33] and PE-Probe [34] may be the closest works to this research. In both works, structural information from PE executables was extracted, e.g. information about how many sections of code were in the executable, the number of standard sections and so on. However, the approach was limited because this type of structural information can be easily changed without affecting the behaviour of the malware.

The obtained results show that building a malware variant detector based on opcode-sequence is feasible. Our prototype NOA achieved a high performance in classifying malware. There has been a huge amount of work for the detection of malware variants in the last years. Besides, there are several considerations regarding the complete viability of our method.

[9] Boosting is a machine-learning technique that builds a strong classifier composed by a high number of weak classifiers [29].

First, the processing overhead of method is highly dependent on the length of the opcode sequences. In our experiments, the impact that the length of opcode sequences has in the time required by the file analysis of this framework has been analyzed. For instance, the system was not able to build the variant malware detector with sequences longer than 3 because the whole experimentation platform could not cope with the process due to the memory requirements. Since the accuracy results using a length of 2 (or even 3) are relatively high, there is no benefit of using such lengths. In addition, a long opcode sequence can be easily evaded by a malware obfuscator through code transposition techniques whereas a short one may be harder to evade.

Second, our representation technique only employs opcodes and discards the operands in the machine code instructions. The work of Bilar [15] studied the ability of single opcodes for representativeness of the legitimacy of an application and it was proved that single opcodes are statistically dependent with regard to the class of a software. However, our representation can be enhanced including operands within the instruction in further work. To use these operands, a classification must be performed first, grouping the operands that have the same meaning together.

Third, because of the static nature of the proposed method, it cannot counter *packed* malware. Packed malware is the result of cyphering the payload of the executable and deciphering it when the executable is finally loaded into memory. Indeed, static detection methods can deal with packed malware only by using the signatures of the packers. Accordingly, dynamic analysis seems to be a more promising solution to this problem [35]. One solution to solve this obvious limitation of our malware detection method is the use of a generic dynamic unpacking schema such as PolyUnpack [36], Renovo [35], OmniUnpack [37] or Eureka [38]. These methods execute the sample in a contained environment and extract the actual payload, allowing further static or dynamic analysis of the executable. Another solution is to use concrete unpacking routines to recover the actual payload, but it requires one routine per packing algorithm [39]. Obviously, this approach is limited to a fixed set of known packers. Likewise, commercial antivirus software also applies *"X-ray"* techniques that can defeat known compression schemes and weak encryption [40]. Still, these techniques cannot cope with the increasing use of packing techniques, and, in our opinion, dynamic unpacking schemes to confront the problem should be used.

Fourth, our method can be considered as a statistical representation of executables. Therefore, an attacker can surpass this method of detection by adding several 'good' opcode sequences. For example, in the field of spam filtering (spam is defined as unsolicited bulk mail), *Good Word Attack* is a method that modifies the term statistics by appending a set of words that are characteristic of legitimate e-mails, thereby bypass spam filters. Nevertheless, this technique can be adapted to this malware detector in some of the methods that have been proposed in order to improve spam filtering, such as *Multiple Instance Learning* (MIL) [41]. MIL will divide an executable or a vector in the traditional methods into several sub-instances and will classify the original vector or classifier based on the sub-instances.

Finally, it may seem that our method detects mainly the compiler used to create executables. In fact, the use of a specific compiler inherently renders an executable rich in several opcode sequences. Nonetheless, when selecting executables to be part of the dataset, they were analyzed using PEiD, a tool that detects most common packers, cryptors and compilers for Portable Executable (PE) files.

After removing the packed ones (as our method would not be able to detect them), there was no significant difference in the compilers used for benign software and malware. It was found that the most common known compilers in the malware dataset were Microsoft Visual Basic, Microsoft Visual C++, Borland Delphi and Borland C++. In the benign dataset, the most common compilers were Microsoft Visual C++, Borland C++ and Borland Delphi. The ability to detect compilers may be applied for our own benefit, to detect whether an executable is packed. If the executable is packed, then the system may unpack it using a dynamic unpacking schema capable of extracting the original payload. Afterwards or if the sample is not packed, the system can analyze it to determine whether it is malware.

6 CONCLUDING REMARKS

In this paper, a new method for malware detection based on information retrieval is described. Opcode sequences were used as the main features for the representation of executables. Through this representation, complete malware detection system was built composed of:

1. a malware database, to store the different representations of variants from several malware families;
2. a system able to generate the representation of the executable under inspection, in other words, a query for the database and
3. a ranking function that computes the similarity between the inspected executable and the malware variants within the database.

This framework was validated using 5 different malware families, using several of their variants to populate the malware database. The remaining ones were used together with a dataset of legitimate applications to test the accuracy of the system. This system achieved, for several experimental configurations, a high detection ratio of malware variants while keeping a low false positive ratio. Besides, the required analysis time was tested and the results showed that the analysis is highly dependent on both opcode sequence length and inspected file size.

The future development of this malware detection tool will be concentrated in three main research areas. First, our work will focus on facing packed executables. Second, the comparison phase will be optimized, reducing the total number of comparisons through a hierarchical search within the database. Finally, the performance overhead of our representation method will be reduced.

REFERENCES

[1] Morley, P.: Processing Virus Collections. In: Proceedings of the Virus Bulletin Conference (VB) 2001, pp. 129–134.

[2] Kuzurin, N.—Shokurov, A.—Varnovsky, N.—Zakharov, V.: In: Concept of Software Obfuscation in Computer Security. Lecture Notes in Computer Science, Vol. 4779, 2007, pp. 281–298.

[3] Bruschi, D.—Martignoni, L.—Monga, M.: Detecting Self-Mutating Malware Using Control-Flow Graph Matching. Lecture Notes in Computer Science, Vol. 4064, 2006, pp. 129–143.

[4] Zhang, Q.—Reeves, D. S.: Metaware: Identifying Metamorphic Malware. In: Proceedings of the 23th Annual Computer Security Applications Conference (ACSAC) 2007, pp. 411–420.

[5] Chouchane, M. R.—Lakhotia, A.: Using Engine Signature to Detect Metamorphic Malware. In: Proceedings of the 4th ACM workshop on Recurring Malcode, pp. 73–78.

[6] Karim, M. E.—Walenstein, A.—Lakhotia, A.—Parida, L.: Malware Phylogeny Generation Using Permutations of Code. Journal in Computer Virology, Vol. 1, 2005, No. 1, pp. 13–23.

[7] Christodorescu, M.—Jha, S.: Testing Malware Detectors. In: ACM SIGSOFT Software Engineering Notes 2004, pp. 34–44.

[8] Sung, A. H.—Xu, J.—Chavez, P.—Mukkamala, S.: Static Analyzer of Vicious Executables (SAVE). In: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC) 2004, pp. 326–334.

[9] Xu, J.—Sung, A.—Chavez, P.—Mukkamala, S.: Polymorphic Malicious Executable Scanner by API Sequence Analysis. In: Proceedings of the 4th International Conference on Hybrid Intelligent Systems 2004, pp. 378–383.

[10] Lo, R. W.—Levitt, K. N.—Olsson, R. A.: MCF: A Malicious Code Filter. Computers & Security, Vol. 14, 1995, No. 6, pp. 541–566.

[11] Bergeron, J.—Debbabi, M.—Erhioui, M. M.—Ktari, B.: Static Analysis of Binary Code to Isolate Malicious Behaviors. In: Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises 1999, pp. 184–189.

[12] Christodorescu, M.—Jha, S.: Static Analysis of Executables to Detect Malicious Patterns. In: Proceedings of the 12th USENIX Security Symposium 2003, pp. 169–186.

[13] Christodorescu, M.—Jha, S.—Seshia, S. A.—Song, D.—Bryant, R. E.: Semantics-Aware Malware Detection. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy, pp. 32–46.

[14] Christodorescu, M.: Behavior-Based Malware Detection. Ph. D. thesis, University of Wisconsin-Madison 2007.

[15] Bilar, D.: Opcodes as Predictor for Malware. International Journal of Electronic Security and Digital Forensics, Vol. 1, 2007, No. 2, pp. 156–168.

[16] Santos, I.—Brezo, F.—Nieves, J.—Penya, Y. K.—Sanz, B.—Laorden, C.—Bringas, P. G.: Opcode-Sequence-Based Malware Detection. In: Proceedings of the 2nd International Symposium on Engineering Secure Software and Systems (ESSoS 2010), Lecture Notes in Computer Science, Vol. 5965, 2010, pp. 35–43.

[17] Baeza-Yates, R. A.—Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA 1999.

[18] Peng, H.—Long, F.—Ding, C.: Feature Selection Based on Mutual Information: Criteria of max-Dependency, max-Relevance, and min-Redundancy. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 27. 2005, pp. 1226–1238.

[19] Mntaras, R. L.: A Distance-Based Attribute Selection Measure for Decision Tree Induction. Machine learning, Vol. 6, 1991, No. 1, pp. 81–92.

[20] Jin, X.—Xu, A.—Bie, R.—Guo, P.: Machine Learning Techniques and chi Square Feature Selection for Cancer Classification Using SAGE Gene Expression Profiles. Lecture Notes in Computer Science, Vol. 3916, 2006, pp. 106–115.

[21] Robertson, S.: Understanding Inverse Document Frequency: On Theoretical Arguments for IDF. Journal of Documentation, Vol. 60, 2004, No. 6, pp. 503–520.

[22] McGill, M. J.—Salton, G.: Introduction to Modern Information Retrieval. McGraw-Hill 1983.

[23] Salton, G.: Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer. Addison-Wesley 1989.

[24] Zhai, C.—Lafferty, J.: A Study of Smoothing Methods for Language Models Applied to Information Retrieval. ACM Transactions on Information Systems, Vol. 22, 2004, No. 2, pp. 179–214.

[25] Chen, P. P. S.: The Entity-Relationship Model Toward a Unified View of Data. ACM Transactions on Database Systems (TODS), Vol. 1, 1976, No. 1, pp. 9–36.

[26] Tata, S.—Patel, J. M.: Estimating the Selectivity of tf-idf Based Cosine Similarity Predicates. ACM SIGMOD Record, Vol. 36, 2007, No. 2, pp .75–80.

[27] Schultz, M. G.—Eskin, E.—Zadok, F.—Stolfo, S. J.: Data Mining Methods for Detection of new Malicious Executables. In: Proceedings of the 22th IEEE Symposium on Security and Privacy 2001, pp. 38–49.

[28] Kolter, J. Z.—Maloof, M. A.: Learning to Detect Malicious Executables in the Wild. In: Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining 2004, pp. 470–478.

[29] Schapire, R. E.: The Boosting Approach to Machine Learning: An Overview. In: Proceedings of the MSRIWorkshop on Nonlinear Estimation and Classification 2003.

[30] Moskovitch, R.—Stopel, D.—Feher, C.—Nissim, N.—Elovici, Y.: Unknown Malcode Detection via Text Categorization and the Imbalance Problem. In: Proceedings of the 6th IEEE International Conference on Intelligence and Security Informatics (ISI) 2008, pp. 156–161.

[31] Zhou, Y.—Inge, W.: Malware Detection Using Adaptive Data Compression. In: Proceedings of the 1st ACM Workshop on AISec 2008, pp. 53–60.

[32] Santos, I.—Penya, Y.—Devesa, J.—Bringas, P. G.: N-Grams-Based File Signatures for Malware Detection. In: Proceedings of the 11th International Conference on Enterprise Information Systems (ICEIS), Vol. AIDSS 2009, pp. 317–320.

[33] Shafiq, M.—Tabish, S.—Mirza, F.—Farooq, M.: PE-Miner: Mining Structural Information to Detect Malicious Executables in Real Time. In: Recent Advances in Intrusion Detection 2009, pp. 121–141.

[34] Shafiq, M.—Tabish, S.—Farooq, M.: PE-Probe: Leveraging Packer Detection and Structural Information to Detect Malicious Portable Executables. In: Virus Bulletin Conference (VB) 2009.

[35] Kang, M.—Poosankam, P.—Yin, H.: Renovo: A Hidden Code Extractor for Packed Executables. In: Proceedings of the 2007 ACM Workshop on Recurring Malcode 2007, pp. 46–53.

[36] Royal, P.—Halpin, M.—Dagon, D.—Edmonds, R.—Lee, W.: Polyunpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In: Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC) 2006, pp. 289–300.

[37] Martignoni, L.—Christodorescu, M.—Jha, S.: Omniunpack: Fast, Generic, and Safe Unpacking of Malware. In: Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC) 2007, pp. 431–441.

[38] Sharif, M.—Yegneswaran, V.—Saidi, H.—Porras, P.—Lee, W.: Eureka: A Framework for Enabling Static Malware Analysis. In: Proceedings of the European Symposium on Research in Computer Security (ESORICS) 2008, pp. 481–500.

[39] Ször, P.: The Art of Computer Virus Research and Defense. Addison-Wesley Professional 2005.

[40] Perriot, F.—Ferrie, P.: Principles and Practise of X-Raying. In: Proceedings of the Virus Bulletin International Conference 2004, pp. 51–66.

[41] Zhou, Y.—Jorgensen, Z.—Inge, M.: Combating Good Word Attacks on Statistical Spam Filters with Multiple Instance Learning. In: Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence 2007, pp. 298–305.

Igor Santos finished his Ph. D. in 2011 with a dissertation about malware detection. He is a researcher in Deustotech, where he conducts research focused mainly in information security, malware detection, content filtering, natural language processing, information retrieval methods, opinion mining, and applied machine learning. He is also lecturer at the Faculty of Engineering of the University of Deusto, where he has taught in undergraduate and postgraduate courses.

Xabier Ugarte-Pedrero finished his studies in computer engineering in 2010 at Deusto University. During 2010–2011, he coursed M. Sc. in information security at the University of Deusto. He joined DeustoTech in September 2010. His research interests include malware unpacking and analysis, natural language processing, software engineering and microbot development and digital electronics. He is currently working towards his Ph. D. dissertation about malware unpacking.

Felix Brezo is a Computer Science and Industrial Management engineer and, currently, a Ph. D. student working at DeustoTech-Computing. His labour as researcher is mainly focused in computer security issues, natural language processing and analytical procedures mainly applying machine learning techniques. His efforts have tackled with malware detection and, more recently, with botnet detection making use of traffic analysis.

Pablo G. Bringas has Ph. D. in computer science and artificial intelligence, M. Sc. in telecommunications, MSc in technical informatics and software engineering. He is currently Head Researcher at DeustoTech – Deusto Technology Foundation, in the S^{3}Lab (Laboratory for Smartness, Semantics and Security), and Assistant Professor at the University of Deusto. He is a member of the Executive Comittee of the Spanish National Consultation Council on Cyber Security.

José María Goméz-Hidalgo has been a lecturer and researcher at the Universidad Complutense de Madrid and the Universidad Europea de Madrid for 16 years, and R & D Director at the Optenet multinational security company. His main research interests include natural language processing and machine learning, text and data mining and information retrieval, with applications to information access in newspapers and biomedicine, and adversarial information retrieval with applications to spam filtering and Web filtering, and children protection on the Web. He is currently the Chairman of the content filtering standardization committee CEN/PC 365.