Computers & Security 95 (2020) 101873

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

HYDRA: A multimodal deep learning framework for malware classification

Daniel Gibert*, Carles Mateu, Jordi Planes

University of Lleida, Jaume II, 69, Lleida, Spain

ARTICLE INFO

Article history:
Received 26 December 2019
Revised 31 March 2020
Accepted 8 May 2020
Available online 12 May 2020

Keywords:
Malware classification
Machine learning
Deep learning
Feature fusion
Multimodal learning

ABSTRACT

While traditional machine learning methods for malware detection largely depend on hand-designed features, which are based on experts' knowledge of the domain, end-to-end learning approaches take the raw executable as input, and try to learn a set of descriptive features from it. Although the latter might behave badly in problems where there are not many data available or where the dataset is imbalanced. In this paper we present HYDRA, a novel framework to address the task of malware detection and classification by combining various types of features to discover the relationships between distinct modalities. Our approach learns from various sources to maximize the benefits of multiple feature types to reflect the characteristics of malware executables. We propose a baseline system that consists of both hand-engineered and end-to-end components to combine the benefits of feature engineering and deep learning so that malware characteristics are effectively represented. An extensive analysis of state-of-the-art methods on the Microsoft Malware Classification Challenge benchmark shows that the proposed solution achieves comparable results to gradient boosting methods in the literature and higher yield in comparison with deep learning approaches.

1. Introduction

In recent years the number, and damage, of cyberattacks has drastically increased, up to the point that cyberthreats are considered among the top most notable risks for the upcoming years. The role cyberwarfare plays in our daily lives should not be underestimated, we have recenlty seen its influence on major elections and on crippling businesses overnight. The most notorious cyberespionage campaign against a political party took place in 2015 and 2016, affecting the Democratic National Committee (DNC), in which hackers infiltrated the DNC computer network and ended up releasing private and confidential information in a collection including approximately 19,000 emails and 8000 attachments from the DNC. Additionally, according to Symantec (Chandrasekar et al., 2017), the number of new ransomware families discovered during 2016 tripled, and they logged a 36% growth in ransomware infections. In 2017 two major cyberattacks, Wannacry in May, followed by Petya in July, held computer systems from all over the globe to ransom. Both malicious programs exploited a vulnerability of Microsoft Windows OS, codenamed EternalBlue (Vulnerabilities and Exposures, 2016), to rapidly spread from one computer to other computers on the same network.

The global malware industry is estimated to be worth millions or even billions of dollars, and continues to grow every year. The underground services market is maturing at increased rates, providing malicious software, cyber-capabilities, and products to other criminals, gangs, and even nation states. It has evolved into a powerful ecosystem, built to exploit every opportunity and weakness in an increasingly connected world. For instance, this year malware developers aimed to mine cryptocurrencies by stealing users' computing power or directly taking the credentials of their cryptocurrency wallet (Cleary et al., 2018; Daniely et al., 2018).

To keep up with malware evolution and be able to reduce the impact of cyberattacks it is necessary to improve computer systems' cyber defenses. One essential defense element is endpoint protection. These defenses range from appropriately keeping up-to-date with patches, to using host-based firewalls against malware. Specifically, anti-malware solutions are the last layer of defense against a cyberattack by preventing, detecting, and removing malicious software. Malware classification approaches can be classified into two categories: (1) static analysis-based detection and (2) dynamic analysis-based detection. On the one hand, static analysis examines the code of a program without executing it. On the contrary, dynamic analysis monitors the behavior of the program in

* Corresponding author.
E-mail addresses: daniel.gibert@diei.udl.cat (D. Gibert), carlesm@diei.udl.cat (C. Mateu), jplanes@diei.udl.cat (J. Planes).

https://doi.org/10.1016/j.cose.2020.101873
0167-4048/

the system. Afterwards, based on the information extracted from both static and dynamic analysis, experts manually define a set of rules to detect current and incoming threats.

Decades ago the number of malware threats was relatively low and simple hand-crafted rules were often enough to detect threats. Lately, however, the massive growth of malware streams does not allow anti-malware solutions to rely solely on expensive hand-designed rules. Consequently, machine learning has become an appealing signature-less approach for detecting and classifying malware due to its ability to generalize in relation to never-before-seen malware. Traditional machine learning approaches rely mainly on feature engineering to extract a set of discriminative features that provide a feature vector representation of malware that a classifier uses to determine the maliciousness of an executable. However, these solutions depend almost entirely on the ability of the domain experts to extract characterizing features that accurately represent the malware. Nevertheless, following recent advances in the machine learning field, there has been a trend towards replacing traditional machine learning pipeline systems with an end-to-end learning algorithm. An end-to-end learning algorithm takes the raw executable as input and tries to directly recognize whether or not it is malicious, or the malware family to which it belongs. Despite the benefits of end-to-end learning, almost no preprocessing and no hand-engineered knowledge, these systems might behave badly in problems where there are not many data available or the dataset is imbalanced.

To mitigate the limitations of end-to-end learning, in this paper we present HYDRA, a novel framework for malware classification using information from many sources to reflect various characteristics of malware executables. To the best of our knowledge, this research is the first application of multimodal deep learning to malware classification. The multimodal learning pipeline combines both hand-engineered and end-to-end components to build a robust classifier. This is achieved by means of a modular architecture that can be broken down into one or more subnetworks, depending on the different types of input of the system. Each of the subnetworks can be either independently trained to solve the same task and then combined, or jointly trained. The features learned by each component are gradually fused into a shared representation layer constructed by merging units with connections coming into this layer from multiple modality-specific paths. To avoid overfitting, during training we randomly drop out the information provided by one or more modalities of information. This prevents the co-adaptation of the subnetworks to a specific feature type. The performance of our multimodal learning algorithm has been evaluated on the Microsoft Malware Classification benchmark. In addition, we provide a comparison with state-of-the-art methods in the literature, including gradient boosting and deep learning methods.

The rest of the paper is organized as follows. Section 2 details the research in the machine learning field to address the problem of malware detection and classification. Section 3 introduces the problem of malware classification and, specifically, the task of classifying malicious Windows executables. Section 4 provides a detailed description of the different types of features or modalities used in our baseline framework. Section 5 presents the architecture of the multimodal neural network and Section 6 describes the experimentation. Lastly, Section 7 summarizes our research and provides future remarks on the ongoing research trends.

2. Related work

Machine learning approaches for tackling the problem of malware detection and classification can be divided into two groups: (1) static approaches (Ahmadi et al., 2016; Yuxin and Siyi, 2017) and (2) dynamic approaches (Bidoki et al., 2017; Ghiasi et al., 2015; Salehi et al., 2017).

Static approaches extract features without involving the execution of malware. Dynamic approaches require the program's execution.

Machine learning approaches are appealing to detect and classify malicious software because of their ability to recognize unseen malware by detecting patterns drawn from previous data. The machine learning workflow involves gathering available data, cleaning/preparing data, building models, validating and deploying in production. The data preparation process in traditional machine learning approaches includes preprocessing the executable, feature extraction, selection and reduction. Afterwards, the remaining features are used to train a model to solve the problem at hand, either to detect malware or to group malware into families. Thus, traditional machine learning methods rely mainly on feature engineering to extract discriminant features from a computer program that provide an abstract view that a classifier uses to make decisions about the inputs. On the contrary, end-to-end learning approaches jointly perform feature extraction and classification, replacing the aforementioned feature engineering process by a fully trainable system. An up-to-date review of machine learning approaches applied to either the problem of malware detection and classification is provided in (Souri and Hosseini, 2018; Ucci et al., 2019).

A description of the most relevant static methods in the literature, divided by the type of input features, is provided below.

The first machine learning classifiers were based on n-gram analysis. An n-gram is a contiguous sequence of n items from a text. In our domain, the items can be byte values (Jain and Meena, 2011; Moskovitch et al., 2008) or assembly language instructions (Santos et al., 2013; Shabtai et al., 2012), depending on the source of information. However, dealing with long n-grams is computationally prohibitive, as the number of unique combinations jointly increases exponentially with N. Consequently, researchers proposed various methods to learn n-gram like signatures without having to enumerate all n-grams during training. Gibert et al. (2017) and McLaughlin et al. (2017) proposed a shallow convolutional neural network architecture to extract n-gram like signatures from a sequence of opcodes to classify Windows and Android malware, respectively. Raff et al. (2018) and Krčál et al. (2018) designed end-to-end systems to learn directly from raw byte inputs, by stacking one or more convolutional layers to learn features from the hexadecimal representation of executables.

Malware authors usually protect malicious software against reverse engineering and detection by using encryption or packing methods to hide the malicious code. Entropy analysis has long been employed to detect the presence of encrypted and packed segments of code, as those segments tend to have higher entropy than native code. For instance, Lyda and Hamrock (2007) minutely examined a corpus of files consisting of plain text files, native, compressed and encrypted executables and noted that the average entropy of the files was 4.347, 5.09, 6.80 and 7.17, respectively. However, malware developers employ more or less sophisticated techniques to bypass simple entropy filters. As a result, researchers started examining what is known as the structural entropy of an executable (Sorokin, 2011). That is, an executable is split into non-overlapping chunks of fixed length and, for each chunk, we measure its entropy. Thus, each file is represented as an entropy time series. Wojnowicz et al. (2016) developed a method to quantify the extent to which variations in a file's structural entropy make it suspicious. In addition, Gibert et al. (2018b) proposed a convolutional neural network-based system to group malware into families.

The file format of the executables is a source of interesting features. In particular, Portable Executable (PE) files have information on the associated dynamically linked libraries, the sections of the

program and their respective sizes, among others. More specifically, the invocation of Application Programming Interface (API) functions and system calls offers information about services and resources provided by the OS, which can be used to model program behavior (Aafer et al., 2013; Sami et al., 2010).

Moreover, it is common to combine information about the API function calls with other types of features in order to build a more robust classifier (Ahmadi et al., 2016; Hassen et al., 2017; Zhang et al., 2016).

An interesting approach is to represent the function calls as a directed graph, known as Function Call Graph (CFG), where its vertices represent the functions a computer program comprehends and the edges symbolize the function calls. For example, Kinable and Kostakis (2011) proposed to clusterizing malware based on the structural similarities between function call graphs, using the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm. Hassen and Chan (2017) proposed a linear time function call graph vector representation for malware classification and showed how to successfully combine the graph features with other non-graph features.

An original way to represent an executable is to reorganize its byte code as a gray scale image (Nataraj et al., 2011), where every byte is interpreted as one pixel in the image, and values range from 0 to 255 (0:black, 255:white). From this representation, it is possible to extract features describing the textures in an image, such as GIST (Nataraj et al., 2011), Haralick (Ahmadi et al., 2016), Local Binary Patterns (Ahmadi et al., 2016) and PCA (Narayanan et al., 2016) features that a classifier can use to classify malware. Additionally, Gibert et al. (2018c) and Khan et al. (2018) evaluated the use of convolutional neural network architectures to detect the presence of specific features and patterns in the data that can be used to group malware into families.

However, using only one type of features is not enough to correctly detect or classify malware in a real-world environment as the obfuscation techniques employed by malware authors might conceal one or more features used by the machine learning model. Thus, efforts are being made to design algorithms that can handle multiple categories of features. Current methods can be divided into two groups, depending on where the features are combined. On the one hand, early or data-level fusion approaches involve the integration of multiple data sources into a single feature vector that is used as input to a machine learning algorithm. For instance, Ahmadi et al. (2016) presented a categorization system that fuses multiple feature types (entropy statistics, image representation, frequency of opcodes, registers, symbols and Windows Application Programming Interfaces) into a single feature vector used to train boosting trees. On the contrary, late or decision-level fusion approaches are those that aggregate the decisions from multiple classifiers, each trained in separate modalities. To illustrate the point, Hassen et al. (2017) proposed an ensemble of individual malware classifiers to precisely classify malware, with a convolutional neural network to process the binary content represented as an image and a feedforward neural network feed with opcode n-gram features as input. As far as we know, there are no approaches in the literature that have successfully tried a deep or intermediate fusion strategy, where all modalities are fused into a single shared representation at some depth or gradually fused, for the task at hand.

3. The task of malware classification

This paper addresses the task of malware classification, which refers to the task of grouping or categorizing malware into families based on its characteristics and behavior. Distinguishing and classifying different types of malware is an important task as it provides information to better understand how the malware has infected the computers or devices, their threat level and how to protect against them. Notice that the only difference between the malware detection and classification tasks is the output of the system implemented. For instance, a malware detection system would receive as input an executable x and would output a single value $y = f(x)$ in the range from 0 to 1, indicating the maliciousness of the executable. A value closer to 0 indicates that the executable is benign and a value closer to 1 indicates that the executable is malicious. On the contrary, a classification system outputs the probability of a given executable belonging to each output category or family. Furthermore, the features extracted from a computer program are useful both for detecting if it is malicious and for classifying it.

Table 1
Class distribution in the microsoft malware classification challenge dataset.

Family name	#Samples	Type
Ramnit	1541	Worm
Lollipop	2478	Adware
Kelihos_ver3	2942	Backdoor
Vundo	475	Trojan
Simda	42	Backdoor
Tracur	751	TrojanDownloader
Kelihos_ver1	398	Backdoor
Obfuscator.ACY	1228	Any kind of obfuscated malware
Gatak	1013	Backdoor

The task of malware detection and classification has not received the same attention in the research community as other applications, where rich labeled datasets exist, including image classification, speech recognition, etc. Due to legal restrictions, benign binaries are not shared, as they are often protected by copyright laws and thus, researchers cannot share the binaries used in their research. On the other hand, malicious binaries are shared through web sites such as VirusShare and VXHeaven. Nevertheless, unlike other domains where data may be labeled very quickly and in many cases by a non-expert, determining whether a file is malicious or its corresponding family or class can be a time-consuming process, even for security experts. Furthermore, services like VirusTotal specifically restrict sharing the vendor anti-malware labels to the public. Thus, for reproducibility purposes, we evaluated the multimodal deep learning system on the data provided by Microsoft for the Big Data Innovators Gathering Challenge (Ronen et al., 2018) of 2015, a high-quality public labeled benchmark. A complete description of the dataset is provided in the next section.

3.1. The microsoft malware classification challenge

Microsoft provided almost half a terabyte of malicious software for the Big Data Innovators Gathering Challenge (Ronen et al., 2018) of 2015. Nowadays, the dataset is hosted on Kaggle[1] and is publicly accessible. The dataset has become the standard benchmark to evaluate machine learning techniques for the task of malware classification. The set of samples represents 9 different malware families, where each sample is identified by a hash and its class, an integer representing one of the 9 malware families to which the malware belongs: (1) Ramnit, (2) Lollipop, (3) Kelihos_ver3, (4) Vundo, (5) Simda, (6) Tracur, (7) Kelihos_ver1, (8) Obfuscator.ACY and (9) Gatak. Fig. 1 displays the distribution of classes of the training data. We can observe that the number of instances of some families significantly outnumbers the number of instances of other families. There are two kinds of repre-

[1] https://www.kaggle.com/c/malware-classification/.

```
00401000 56 8D 44 24 08 50 8B F1 E8 1C 1B 00 00 C7 06 08
00401010 BB 42 00 8B C6 5E C2 04 00 CC CC CC CC CC CC CC
00401020 C7 01 08 BB 42 00 E9 26 1C 00 00 CC CC CC CC CC
00401030 56 8B F1 C7 06 08 BB 42 00 E8 13 1C 00 00 F6 44
00401040 24 08 01 74 09 56 E8 6C 1E 00 00 83 C4 04 8B C6
00401050 5E C2 04 00 CC CC CC CC CC CC CC CC CC CC CC CC
00401060 8B 44 24 08 8A 08 8B 54 24 04 88 0A C3 CC CC CC
00401070 8B 44 24 04 8D 50 01 8A 08 40 84 C9 75 F9 2B C2
00401080 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
00401090 8B 44 24 10 8B 4C 24 0C 8B 54 24 08 56 8B 74 24
004010A0 08 50 51 52 56 E8 18 1E 00 00 83 C4 10 8B C6 5E
004010B0 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
004010C0 8B 44 24 10 8B 4C 24 0C 8B 54 24 08 56 8B 74 24
004010D0 08 50 51 52 56 E8 65 1E 00 00 83 C4 10 8B C6 5E
004010E0 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
004010F0 33 C0 C2 10 00 CC CC CC CC CC CC CC CC CC CC CC
00401100 B8 08 00 00 00 C2 04 00 CC CC CC CC CC CC CC CC
00401110 B8 03 00 00 00 C3 CC CC CC CC CC CC CC CC CC CC
00401120 B8 08 00 00 00 C3 CC CC CC CC CC CC CC CC CC CC
00401130 8B 44 24 04 A3 AC 49 52 00 B8 FE FF FF FF C2 04
00401140 00 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
00401150 A1 AC 49 52 00 85 C0 74 16 8B 4C 24 08 8B 54 24
00401160 04 51 52 FF D0 C7 05 AC 49 52 00 00 00 00 00 B8
00401170 FB FF FF FF C2 08 00 CC CC CC CC CC CC CC CC CC
00401180 6A 04 68 00 10 00 00 68 68 BE 1C 00 6A 00 FF 15
00401190 9C 63 52 00 50 FF 15 C8 63 52 00 8B 4C 24 04 6A
```

Fig. 1. Hex view.

sentations when speaking about binary executables. On the one hand, an executable can be represented as a sequence of hexadecimal values for corresponding bytes of a binary file. For instance, approaches (Gibert et al., 2018b; Nataraj et al., 2011; Wojnowicz et al., 2016) are based on features extracted from this hexadecimal representation. On the other hand, the content of a binary file can be reverted/translated to assembly language. This process is known as disassembly. Common disassemblers are IDA Pro or Radare2. Approaches (Gibert et al., 2017; Kinable and Kostakis, 2011; Sami et al., 2010) illustrate this point.

3.1.1. Hexadecimal representation

The hex view represents the machine code as a sequence of hexadecimal digits. See Fig. 1. Each line is composed of the starting address of the machine codes in the memory and an accumulation of consecutive 16 byte values.

From this kind of representation one can extract byte n-grams, calculate the structural entropy of an executable, represent its binary content as a gray scale image, etc.

3.1.2. Assembly language source code

The assembly language source code contains the symbolic machine code of the executable as well as metadata information such as rudimentary function calls, memory allocation and variable information. A snapshot of a piece of one assembly file is shown in Fig. 2.

Assembly language consists of three types of statements:

1. Instructions or assembly language statements. An instruction defines the operation to execute. Instructions are entered one instruction per line. Their format is as follows:
   `[label] mnemonic [operands]`
   An instruction is composed of two parts: (1) the name of the instruction to be executed and (2) the operands or parameters of the command.
   ```
   INC COUNT
   MOV TOTAL, 48
   ```
2. Assembler directives or pseudo-ops. Assembler directives are the commands part of the assembly syntax but not related to the processor instruction set.
3. Macros. A macro is a sequence of instructions assigned by a name that could be used anywhere in the program.
   ```
   %macro macro_name num_params
   <macro body>
   % endmacro
   ```

4. Modalities description

This paper proposes a multimodal deep learning system to categorize malware into families that involves multiple modalities of data:

1. The list of Windows API functions calls.
2. The sequence of assembly language instructions representing malware's assembly language source code.
3. The sequence of hexadecimal values representing malware's binary content.

These feature types have been chosen because of their respective advantages and limitations. A detailed description and an in-depth anlysis of the aforementioned modalities are provided below, together with the definition of the individual components of the multimodal architecture.

```
.text:00401081 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC     align 10h
.text:00401090 8B 44 24 10                                        mov     eax, [esp+10h]
.text:00401094 8B 4C 24 0C                                        mov     ecx, [esp+0Ch]
.text:00401098 8B 54 24 08                                        mov     edx, [esp+8]
.text:0040109C 56                                                 push    esi
.text:0040109D 8B 74 24 08                                        mov     esi, [esp+8]
.text:004010A1 50                                                 push    eax
.text:004010A2 51                                                 push    ecx
.text:004010A3 52                                                 push    edx
.text:004010A4 56                                                 push    esi
.text:004010A5 E8 18 1E 00 00                                     call    _memcpy_s
.text:004010AA 83 C4 10                                           add     esp, 10h
.text:004010AD 8B C6                                              mov     eax, esi
.text:004010AF 5E                                                 pop     esi
.text:004010B0 C3                                                 retn
.text:004010B0                                                   ; ----------------------
.text:004010B1 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC     align 10h
.text:004010C0 8B 44 24 10                                        mov     eax, [esp+10h]
.text:004010C4 8B 4C 24 0C                                        mov     ecx, [esp+0Ch]
.text:004010C8 8B 54 24 08                                        mov     edx, [esp+8]
.text:004010CC 56                                                 push    esi
.text:004010CD 8B 74 24 08                                        mov     esi, [esp+8]
.text:004010D1 50                                                 push    eax
.text:004010D2 51                                                 push    ecx
.text:004010D3 52                                                 push    edx
.text:004010D4 56                                                 push    esi
.text:004010D5 E8 65 1E 00 00                                     call    _memmove_s
.text:004010DA 83 C4 10                                           add     esp, 10h
.text:004010DD 8B C6                                              mov     eax, esi
.text:004010DF 5E                                                 pop     esi
.text:004010E0 C3                                                 retn
.text:004010E0                                                   ; ----------------------
.text:004010E1 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC     align 10h
.text:004010F0 33 C0                                              xor     eax, eax
.text:004010F2 C2 10 00                                           retn    10h
.text:004010F2                                                   ; ----------------------
```

Fig. 2. Assembly view.

4.1. Windows API function calls

The frequency of use of Application Programming Interfaces (API) and their function calls are regarded as very characterizing features. Literature has demonstrated that API calls can be analyzed to model the program behavior. API functions and system calls are related with services provided by operating systems. They support various key operations such as networks, security, system services, file management, and so on. In addition, they include various functions to utilize system resources, such as memory, file system, network or graphics. There is no other way for software to access system resources that are managed by operating systems without using API functions or system calls and thus, API function calls can provide key information to represent the behavior of the software. In this work, each API function and system call is treated as a feature. The feature range is [0,1]; 0 (or False) if the API function or system call hasn't been invoked by the program; 1 (or True) otherwise. Alternatively, one can count how many times each API function has been called during the execution of the program.

Because many malware programs are packed, leaving only the stub of the import table or perhaps even no import table at all, our approach will search for the name of the dynamic link library or function in the body of the suspected malware (by disassembling the executable).

The number of Windows OS API functions is extremely large. Considering all of them would bring little or no meaningful information for malware classification. Consequently, the analysis was restricted to a subset of API functions. The complete list of Windows API functions was reduced to only those functions that were invoked at least thrice in our training data. The remaining functions were not considered in the analysis. Among the most used functions we found the following: the *Sleep* function, which is used to evade dynamic analyzers, the *VirtualAlloc* function, which is used to allocate memory to store the unpacked code in the newly allocated block and perform a jump to run the code from there, and the *LoadLibraryA* and *GetProcAddress* functions, which are both used to resolve the addresses of the API calls made by the program.

The total number of functions invoked is 10670, almost equal to the number of training samples, which might lead to overfitting. High dimensionality results in increased cost and complexity for both feature extraction and classification. In practice, the algorithm might perform badly if the dimensionality is increased beyond a certain point when there is a finite number of training samples. This problem is known as the curse of dimensionality. Consequently, feature selection has been applied to select only a subset of the features. Feature selection is the process of selecting a subset of features that are more relevant to a predictive modeling problem.

This helps to remove unneeded, irrelevant and redundant attributes from the data that do not contribute to the accuracy of a predictive model. Afterwards, the subset of features is used to learn the predictive modeling problem. An overview of the proposed method is presented in Fig. 3. In our specific implementation, the classification algorithm is a feed-forward network whose hyper-parameters have been selected using a grid search. The experiments to select the optimal subset of features are described in detail in Section 6.1. A detailed description of our feed-forward network architecture is introduced below.

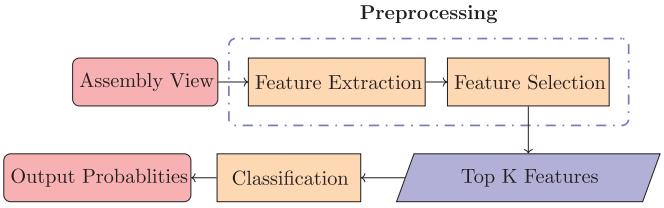

Fig. 3. Traditional pipeline of an API-based malware classification system.

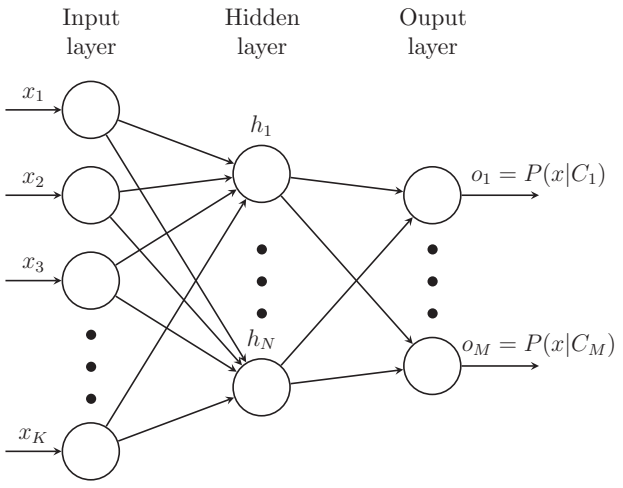

Fig. 4. API-based feed-forward network architecture. *N* and *M* are equal to 250 and 9, respectively.

4.1.1. API-Based feed-forward neural network

An overview of the architecture is described in Fig. 4. The input of the network is a vector containing the K most relevant API-based features according the feature selection technique of our choice. The output of the network is given by the function $f(x) = f^{(2)}(f^{(1)}(x))$, where $f^{(1)}$ refers to the first layer or hidden layer of the network and $f^{(2)}$ refers to the output layer. The mathematical formulation of $f^{(i)}$ is $f^{(i)} = \alpha(Wx + b)$, where α represents the activation function, x the input of the layer and W and b the weights and biases, respectively. In particular, the activation function of the hidden layer is the ELU function (Clevert et al., 2015) while the output layer has no activation function. Instead, it calculates the softmax function to generate the normalized output probabilities.

4.2. Mnemonics analysis

The most common approach to categorize a given sample of text in natural language processing is through n-gram analysis. N-gram analysis calculates the n-gram words distribution of a file as a means of solving a predictive modeling problem. In addition, n-grams have been one of the most popular features used for malware detection or classification (Santos et al., 2013; Shabtai et al., 2012). The simplest approach is to capture only the instruction used as the base. On encountering the instruction *mov eax, [esp+10h]* we simply reduce it to *mov*.

Specifically, mnemonic n-grams are extracted from the sequence of mnemonics included in the assembly language source code of malware. To give a specific example of the process, the mnemonics sequence in Fig. 2, from bytes 00,401,090 to 004010B0 would have the following 2-grams:

```
[[mov,mov], [mov,mov], [mov,push],
[push,mov],
[mov,push], [push,push], [push,push], [push,
push],
```

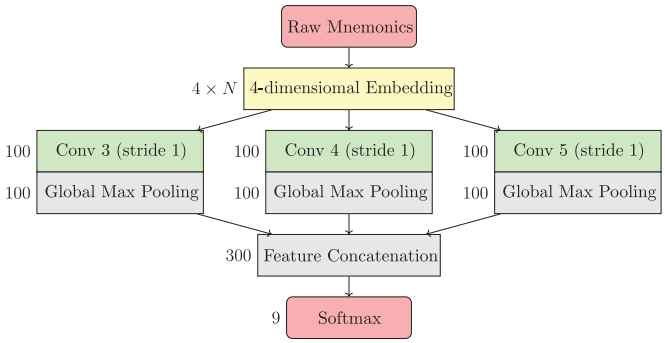

Fig. 5. Convolutional neural network for malware mlassification from sequences of mnemonics.

```
[push,call], [call,add], [add,mov],
[mov,pop],
[pop, retn]]
```

N-gram based methods construct a feature vector representation of malware where each element in the vector indicates the number of appearances of a particular n-gram in the instruction sequence.

The main drawback of n-gram based methods is that the number of unique n-grams depends on n, the number of mnemonics that each n-gram will contain. Since the number of unique n-grams is huge, it is difficult to run machine learning algorithms on the original data. One solution is to perform feature selection, i.e. a process of identifying the best features and filtering out less important features. Another solution was proposed by (Gibert et al., 2017), who presented an alternative to n-gram counts using convolutional neural networks to automatically learn the most discriminative features from a sequence of mnemonics without having to apply any feature selection technique to make the problem tractable.

4.2.1. Convolutional neural network as an alternative to N-grams

The main advantage of a convolutional neural network based approach is that it removes the need to manually enumerate the large number of n-grams during training, as n-gram based approaches do. Instead, it learns n-gram like signatures through the convolutional layers. The most notable implication of such an approach is the elimination of the traditional pipeline composed of feature extraction, feature selection and reduction and classification, as both procedures are optimized together during training.

Due to the advantages of a convolutional neural network based approach, the component responsible for addressing this modality of data has been constructed considering the architecture presented by Gibert et al. (2017) as baseline, with a few minor modifications. The network differs on: (i) the kernel sizes, (ii) the number of filters per size, and (iii) the size of the input layer. In addition, ours also has dropout applied to the input layer. The overall architecture is presented in Fig. 5. It comprises the following layers:

- **Input layer.** The network takes as input an executable represented as a sequence of mnemonics. As the network cannot be fed with text just as strings, each mnemonic is converted to a one-hot vector. To form a one-hot vector, we associate each mnemonic with a numerical ID in the range 1 to I, where I is the vocabulary size. A one-hot vector is a vector of zeros of size I, with a '1' in the position of the mnemonics' ID.
- **Embedding layer.** One-hot vectors cannot encode semantic information about similar operations or similar meaning. To address this issue, each mnemonic is represented as a low-dimensional vector of real values (word embedding) of size

M, where each value captures a dimension of the mnemonics' meaning.

Convolutional layer. This layer is responsible for convolving various filters over the mnemonic sequences and extracting the n-gram like features from it. The size of each filter is $h \times k$ where $h \in \{3, 4, 5\}$ and k is equal to the size of the mnemonic's embedding. Consequently, filters are applied to sequences containing from 3 to 5 mnemonics. The activation function adopted is the Exponential Linear Unit or ELU (Clevert et al., 2015). Having different filter' sizes allows the network to detect salient sub-sequences in the sequence of instructions that have variations in size.

Global max-pooling layer. The global max-pooling is applied to extract the maximum activation of each of the feature map activations passed from the convolutional layer.

Softmax layer. It linearly combines the features learned by the previous layers and applies the softmax function to output the normalized probability distribution over malware families.

Xavier's initialization (Glorot and Bengio, 2010) has been used to initialize the weights of the network with the exception of the embedding layer, in which the initial values of the embedding had been initialized with random values from a uniform distribution ranging from -1 to 1. Additionally, dropout (Hinton et al., 2012) was applied to the input, the convolutional and the output layers, with a percentage of 0.1, 0.1 and 0.5 dropped neurons. For a complete description of the experiments performed see Section 6.2.

4.3. Byte analysis

Similar to the mnemonics analysis counterpart, there had been attempts in the literature (Krčál et al., 2018; Raff et al., 2018) to build end-to-end malware detection systems from raw byte sequences. These approaches take as input the raw byte sequences from the hexadecimal representation of the malware's binary content and try to identify whether or not the executable is malicious. The main challenges that these approaches have to deal with are:

- The meaning of any byte is context-dependent and could encode any type of information such as binary code, human-readable text, images, etc. In addition, the same instruction can be encoded using different byte codes depending on its arguments such as the cmp instruction, whose binary code can begin with 0x3C, 0x3D, 0x3A, 0x3B, 0x80, 0x81, 0x38 or 0x39 depending on the arguments given.
- The content of a Portable Executable file exhibits various levels of spatial correlation. Nearby instructions in a function are spatially correlated. However, function calls and jump instructions produce discontinuities over code instructions and functions. Subsequently, these discontinuities are maintained through the binary content.
- By treating an executable as a sequence of bytes, we are dealing with sequences of millions of time steps, becoming one of the most challenging sequence classification problems with regard to the size of the time series.

4.3.1. State-of-the-art methods

Raff et al. (2018) presented a shallow convolutional neural network architecture consisting of an embedding layer, followed by a gated convolutional layer with filters of size 500 combined with a stride of 500, plus a global max-pooling layer and a softmax layer. This architecture will be called MalConv from now on. Cf. Fig. 6.

Krčál et al. (2018) proposed a deeper architecture that comprises an embedding layer followed by four convolutions with strides and max-pooling between the second and third convolutions. Afterwards, global average pooling is applied to generate the average features in the byte sequences. Finally, the features are non-linearly combined through various fully-connected layers and lastly a softmax layer. This architecture will be called DeepConv from now on. Cf. Fig. 7.

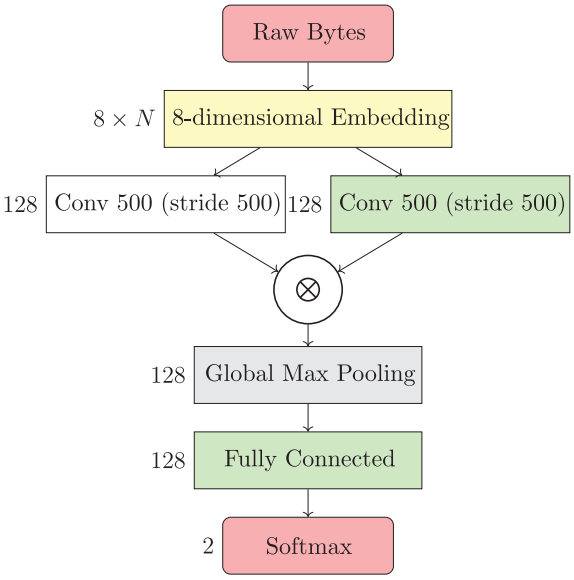

Fig. 6. MalConv architecture.

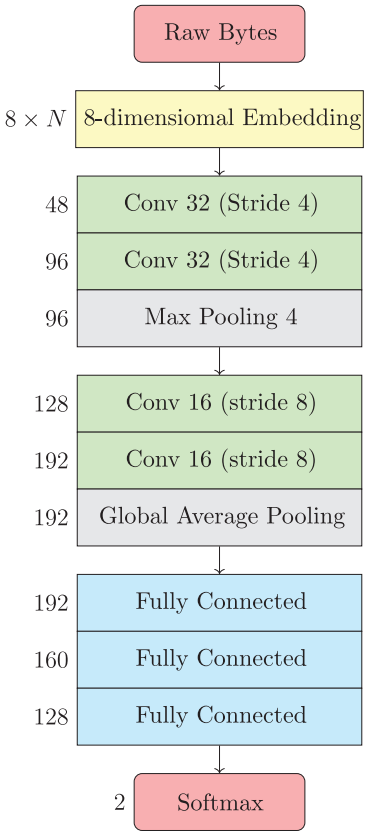

Fig. 7. DeepConv architecture.

Performing convolutions on raw byte values implies interpreting that certain byte values are intrinsically closer to each other than others, which is known to be false, as the meaning of a particular byte is context-dependent. Consequently, both approaches represent bytes as a distributed vector representation of size K,

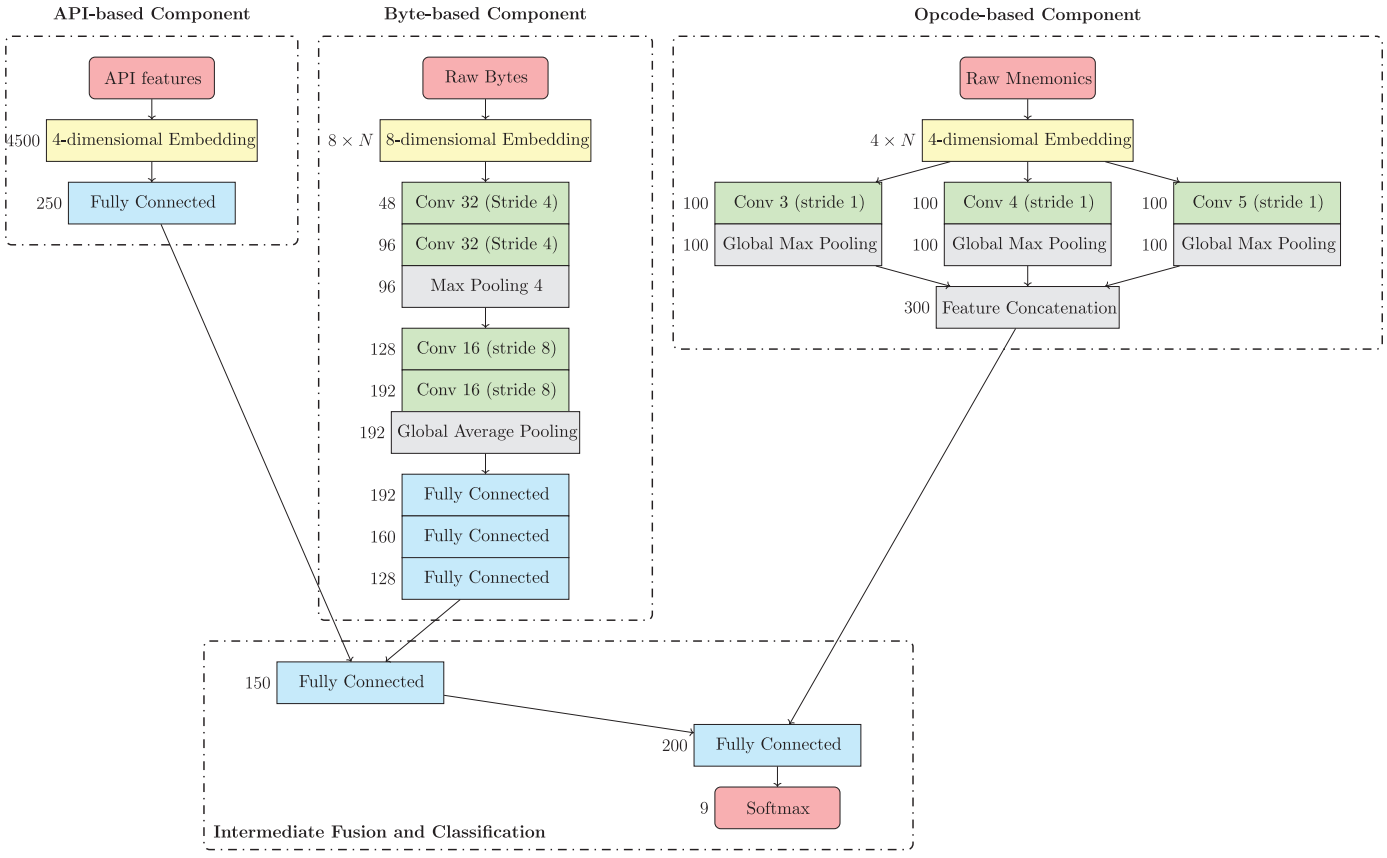

Fig. 8. Multimodal deep neural network architecture.

where each element contributes to the definition of each byte with the goal of capturing the context of a byte in the executable, its semantic similarity as well as its relation with other bytes.

The third component of the multimodal network is responsible for extracting features from the byte sequence representation, which has been constructed according to the network architecture presented by Krčál et al. (2018) as it achieves better performance than MalConv (Raff et al., 2018) in our experiments. See Section 6.3.

5. Multimodal deep learning framework

Fig. 8 shows the architecture of HYDRA, our multimodal deep learning framework for malware classification. This architecture aims to serve as a baseline for future implementations of malware detection systems. It only includes those feature types that the authors consider essential to any machine learning system for the task at hand. Nonetheless, new feature types can be added to the architecture as needed. It consists of 4 main components: (1) the API-based component, (2) the mnemonics-based component, (3) the byte-based component and (4) the feature fusion and classification component. The first three components are not connected to each other. Each one extracts features from a different abstract representation of malware (from a different data modality). The final component is responsible for fusing the features learned by each component into a shared representation and for producing the classification predictions. This architecture consists of both hand-engineered and end-to-end components. On the one hand, the hand-engineered component learns the complex relationships among the input API feature vector. On the contrary, the end-to-end components learn features from malware represented as a sequence of mnemonics and as a sequence of bytes. As end-to-end learning requires a great deal of labeled data to work properly, combining hand-engineered and end-to-end components helps to mitigate the limitations of the system and more specifically, of the byte-based component (See Section 6.3), and to build a stronger classifier by combining the strengths of both approaches.

5.1. Architecture

In this subsection, the architecture of the multimodal neural network is described, in particular the input layer, the three feature components and the fusing component.

Input layer. The dataset consists of a set of pairs x^i, y^i, where x^i is an executable and y^i is the category label or family to which it belongs. Each executable x^i is represented as an n-tuple, where n is equal to the number of different modalities of data. In our case $n = 3$, as each executable is represented as a list of API function calls x^i_A, a sequence of mnemonics x^i_M, and lastly a sequence of bytes x^i_B.

API-based component. Let $x^i_A \in \mathbb{Z}^K$ be the API feature vector of the ith executable in the training set. Each vector consists of K feature values, where each feature indicates whether or not a particular API function has been invoked. For instance, if $x^i_A(j)$ is equal to 1 it indicates that the ith executable has invocked API function j.

The API-based component takes as input the API feature vector and non-linearly combines the features into a low-dimensional feature vector A of size 250, where $A^i = h(x^i_A) = \sigma(W_A x^i_A + b_A)$ and W_A and b_A denote the weights and biases of the fully-connected layer.

Mnemonics-based component. Let x^i_M denote the sequence of mnemonics extracted from the assembly language source

code of the ith executable. As already explained in Section 4.2.1, each mnemonic is mapped to a vector of real numbers of size 4. The mnemonics-based component outputs a feature vector M of size equal to 300 consisting of the concatenation of the 3-gram, 4-gram and 5-gram like signatures.

Byte-based component. Let x_B^i refers to the sequence of bytes extracted from the hexadecimal representation of the binary content of the ith executable. This component outputs a feature vector B containing a low-dimensional representation of the malware's binary content of size 128 (See Section 4.3.1).

Intermediate fusion and classification. The learned representations A, M and B of malware are merged iteratively across multiple fusion layers during training and combined into a shared multimodal representation. This process is known as intermediate fusion. First, vectors A and B are fused into vector C of size 150. Afterwards, vectors C and M are combined in a 1-D vector of size 200, called P. This joint multimodal representation P is later used to classify a malicious executable into the corresponding family, as follows:

$$p = softmax(W_c P + b_c)$$

where p is a vector of size C ($C = 9$), and W_c and b_c are the weights and biases of the layer. The softmax function outputs the probability of an executable belonging to any of the malware families in the training set. The size of vectors C and P were defined during the configuration of the network. Various values for the number of hidden units were tried and the ones yielding to better results were selected. In addition, fusing all features in the same layer yield to worse results even if these are statistically insignificant.

For consistency, all layers have been initialized using Xavier's weight initialization (Glorot and Bengio, 2010). The non-linear functions through all convolutional and fully connected layers are the ELU (Clevert et al., 2015) and the SELU (Klambauer et al., 2017) activation function.

Two aspects have been critical for the success of our multimodal setting: (1) per-modality pretraining and transfer learning, and (2) multimodal dropout.

5.2. Pretraining

In our experiments we observed that taking all modalities of information as input is suboptimal, since it leads to overfitting one subset of features belonging to one modality and underfitting the features belonging to the others. This issue has been addressed by separately pretraining each component and optimizing their hyperparameters for each subtask. Consequently, we randomly split the training data into two sets, 80% for training and 20% for the validation set and we trained three models to classify malware, one model taking one modality as input. Afterwards, the weights of each component in the multimodal neural network are initialized with the optimal pretrained weights that each network has learned for each task. The idea is to transfer the knowledge learned by each model into the multimodal neural network to save training time and help the network converge faster.

5.3. Regularization mechanisms

In a real-world scenario, although malicious and benign executables are given to be analyzed, it is not guaranteed that all of the features can be extracted from the given executables. The only modality that would always be available is the byte sequence. On the contrary, due to encryption and compression, the API function calls and the sequence of assembly language instructions might not be properly retrieved. For instance, there are some samples in the training set that have not been disassembled correctly or could not be disassembled and, consequently, their corresponding assembly language source file contains almost no instructions or no instructions at all (Hu et al., 2016). Thus, we have addressed this issue using modality dropout, which makes the network less sensitive to the loss of one or more channels of information. Modality dropout randomly drops one or more data modalities during training. Additionally, dropout has been applied to both fully-connected and convolutional layers, with a dropout rate equal to 0.5 and 0.1, respectively.

6. Evaluation

We deployed the proposed framework on a machine with an Intel Core i7-7700k CPU, 4xGeforce GTX 1080Ti GPUs and 64 Gb RAM. The GPUs are critical to accelerate the multimodal neural network algorithm. The modules of the framework and the machine learning algorithm have been implemented in Python and Tensorflow (Abadi et al., 2015). Due to the memory resource limitations we have reduced the mini-batch size to 8.

The generalization performance of our approach has been estimated using 10-fold cross validation. Two baseline classifiers were implemented, the Random Guess classifier and the Zero Rule classifier. The accuracy of a Random Guess classifier is calculated as follows:

$$acc = \frac{\sum_{i=1}^{c} p_i n_i}{\sum n_i}$$

where p_i is the probability to say "it is in the ith class" and n_i is the number of samples of class i. Thus, the accuracy of the Random Guess classifier is 0.1755. On the other hand, the Zero Rule classifier it simply outputs the majority class in the dataset. In particular, the accuracy of the Zero Rule classifier is $2942/10,868 = 0.2707$.

Instead of evaluating the model with accuracy alone, we selected the best model according to the macro F1-score. This is because accuracy can be a misleading measure in datasets were there exist a large class imbalance. For instance, a model can correctly predict the value of the majority class for all predictions and achieve a high classification accuracy while making mistakes on the minority and critical classes. The macro F1-score metric penalizes this kind of behavior by calculating the metrics for each label and finding their unweighted mean.

6.1. API-based component performance

Below is an in-depth analysis of the performance of various baseline algorithms to classify malware based on the use of Windows API function calls found in the assembly language source code. In particular,Tables 2–4 provide the 10-fold cross validation accuracy of various algorithms for different K values, where K refers to the K top features selected by either the $\tilde{\chi}^2$ or ANOVA-F feature selection algorithms. The baseline algorithms used in the experiment are logistic regression, support vector machines with linear or rbf kernel, random forests and lastly, gradient boosting. Table 2 presents their performance using as input a feature vector of 0s and 1s of size K, where K refers to the top K features according to the $\tilde{\chi}^2$ score and each value represents whether or not a particular API function has been invoked by the program. Table 3 shows the performance of the algorithms taking as input a feature vector of size K (top K features according to the $\tilde{\chi}^2$ score), where each value indicates the number of times a particular API function has been invoked. Table 4 presents the performance of the baseline algorithms taking as input a feature vector of size K, where K refers to the top K features according to the ANOVA-F

Table 2
$\tilde{\chi}^2$: 10-fold cross validation accuracy of baseline methods - API function call (yes or no).

10-fold cross validation accuracy					
K	Logistic regression	SVM (linear kernel)	SVN (RBF kernel)	Random forests	Gradient boosting
50	0.9004	0.9073	0.8989	0.9351	0.9253
100	0.9519	0.9565	0.9348	0.9623	0.9614
250	0.9640	0.9698	0.9404	0.9717	0.9719
500	0.9688	0.9727	**0.9409**	0.9736	**0.9733**
1000	0.9728	0.9746	0.9380	0.9721	**0.9733**
1500	0.9757	0.9760	0.9376	0.9733	0.9721
2000	0.9758	0.9753	0.9343	**0.9741**	0.9721
2500	0.9761	0.9754	0.9314	0.9728	0.9721
3000	0.9765	0.9758	0.9264	0.9731	0.9718
3500	0.9776	0.9756	0.9234	0.9731	0.9722
4000	0.9784	0.9768	0.9199	0.9723	0.9722
4500	**0.9794**	**0.9774**	0.9173	0.9730	0.9728
5000	**0.9794**	**0.9774**	0.9171	0.9738	0.9717
7000	0.9785	0.9773	0.8954	0.9716	0.9707
10,000	0.9786	0.9773	0.8750	0.9700	0.9709

Table 3
$\tilde{\chi}^2$: 10-fold cross validation accuracy of baseline methods - API function counts.

10-fold cross validation accuracy					
K	Logistic regression	SVM (linear kernel)	SVN (RBF kernel)	Random forests	Gradient boosting
50	0.8535	0.8526	0.7649	0.9414	0.9355
100	0.8954	0.8869	**0.7740**	0.9698	0.9693
250	0.9311	0.9255	0.7405	0.9760	0.9772
500	0.9446	0.9417	0.7152	**0.9780**	0.9767
1000	0.9584	0.9602	0.6814	**0.9780**	0.9764
1500	0.9678	0.9642	0.6539	0.9765	**0.9774**
2000	0.9699	0.9659	0.6472	0.9772	0.9761
2500	0.9706	0.9700	0.6227	0.9769	0.9752
3000	0.9730	0.9720	0.6051	0.9765	0.9760
3500	0.9727	0.9727	0.5951	0.9765	0.9759
4000	0.9729	**0.9729**	0.5870	0.9774	0.9759
4500	0.9736	**0.9729**	0.5791	0.9765	0.9762
5000	**0.9739**	0.9722	0.5212	0.9762	0.9760
7000	0.9737	0.9719	0.4926	0.9762	0.9748
10,000	0.9731	0.9719	0.4554	0.9755	0.9746

Table 4
ANOVA-F: 10-fold cross validation accuracy of baseline methods - API function counts.

10-fold cross validation accuracy					
K	Logistic regression	SVM (linear kernel)	SVN (RBF kernel)	Random forests	Gradient boosting
50	0.9154	0.9106	0.7780	0.9470	0.9398
100	0.9520	0.9357	0.8114	0.9694	0.9671
250	0.9686	0.9520	**0.8155**	0.9758	0.9754
500	0.9689	0.9590	0.7386	0.9753	0.9745
1000	0.9687	0.9634	0.6810	0.9757	0.9741
1500	0.9692	0.9651	0.6555	0.9755	0.9743
2000	0.9698	0.9671	0.6483	0.9761	0.9743
2500	0.9710	0.9679	0.6247	0.9757	0.9738
3000	0.9717	0.9681	0.6053	0.9752	0.9749
3500	**0.9741**	0.9733	0.5948	0.9765	0.9761
4000	0.9734	0.9727	0.5868	0.9761	**0.9764**
4500	0.9739	**0.9734**	0.5790	0.9768	0.9757
5000	0.9737	0.9728	0.5214	**0.9771**	**0.9764**
7000	0.9734	0.9720	0.4927	0.9756	0.9755
10,000	0.9729	0.9719	0.4554	0.9751	0.9752

metric, and each value indicates how many times a particular API function has been called.

According to the empirical observation of Tables 2–4 it can be stated that the $\tilde{\chi}^2$ feature selection metric selects better features than the ANOVA-F measure, as on average all baseline algorithms trained on the subset of features retrieved by the $\tilde{\chi}^2$ metric achieved higher accuracies. Additionally, it can be observed that the highest accuracy was reached by the logistic regression algorithm having as input the top 4500 features ranked with the $\tilde{\chi}^2$ feature selection metric. Consequently, this subset has been used to train the component of the multimodal network responsible for classifying malware based on the Windows API function calls.

The optimal architecture of the API-based component consists of only one hidden layer with 250 units. This configuration was selected according to a grid search over the hyperparameters of the network (Table 5). All models were trained using dropout in both input and hidden layers of 0.1 and 0.5. We can observe that increasing the number of units in the hidden layers or the num-

Table 5
Feed-forward neural network grid search. The architecture of a feed forward neural network is defined as follows: NN $F \times H_1 \times H_2$, where F is the size of the input feature vector, H_1 and H_2 is the number of neurons in the first and second hidden layers, respectively.

Algorithm	Accuracy	Macro F1-score
Random guess	0.1755	–
Zero rule	0.2707	–
Logistic regression	0.9810	0.9573
NN 4500 × 30	0.9824	0.9566
NN 4500 × 50	0.9826	0.9570
NN 4500 × 100	0.9829	0.9602
NN 4500 × 250	**0.9833**	**0.9621**
NN 4500 × 500	0.9829	0.9617
NN 4500 × 2000	0.9828	0.9602
NN 4500 × 2500	0.9825	0.9603
NN 4500 × 30 × 20	0.9823	0.9612
NN 4500 × 50 × 20	0.9822	0.9564
NN 4500 × 50 × 30	0.9824	0.9580
NN 4500 × 1000 × 100	0.9820	0.9603

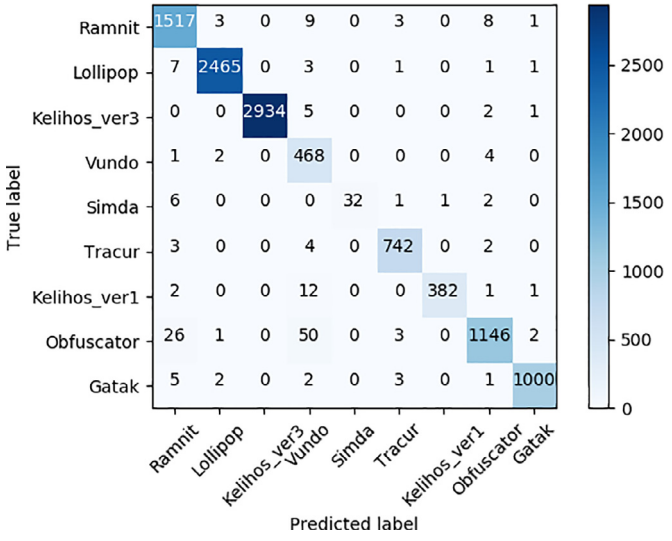

Fig. 9. NN 4500x250 confusion matrix.

ber of layers does not significantly improve the performance of the model and in particular, the highest accuracy and macro F1-score was achieved by a network of only 250 units.

Fig. 9 shows the confusion matrix of the 10-fold cross validation procedure. The major source of errors comes from the misclassification of samples belonging to the Obfuscator.acy family. In particular, 82 out of the 1228 Obfuscator.acy samples have been incorrectly classified.

6.2. Mnemonic-based component performance

Previously to training, the vocabulary (number of distinct mnemonics) was reduced to only consist of those mnemonics that appeared in at least three different executables. Those mnemonics that appeared less than three times in the training set were converted to the UNK token.

Initializing the word vectors with those vectors learned from an unsupervised learning model it has been a common practice in the literature (Collobert et al., 2011), as it improves the performance in tasks where there is no large training set available. Consequently, we initialized the mnemonic vectors using vectors of dimensionality 4 trained using either CBOW or Skip-Gram architecture (Mikolov et al., 2013). However, we did not observe any relevant improvement in either case. Figs. 12 and 11 show the confusion matrices for such architectures.

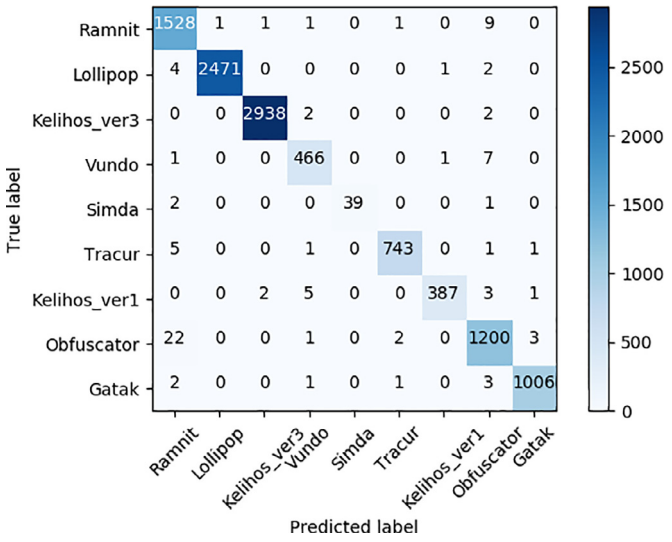

Fig. 10. CNN-rand confusion matrix.

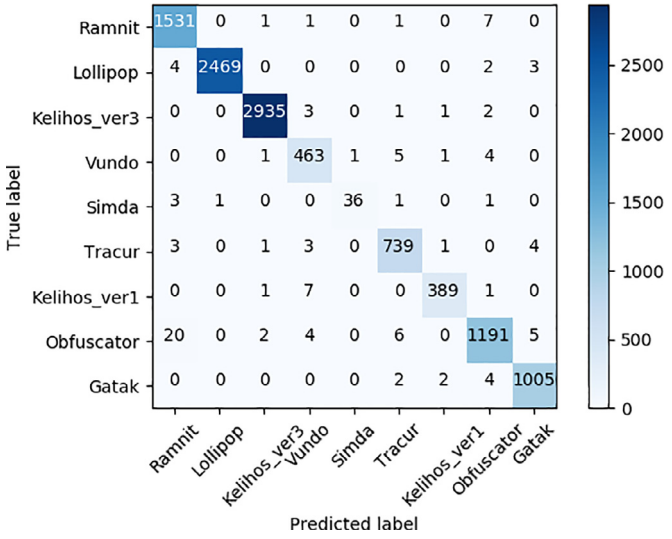

Fig. 11. CNN-skipgram confusion matrix.

In the next experiment, we test three different initialization settings for the mnemonic embeddings by computing the 10-fold cross validation accuracy and macro F1-score achieved by our convolutional neural network. The three settings are the following:

- CNN-rand. Baseline model where all mnemonic vectors are randomly initialized and then modified during training. See Fig. 10.
- CNN-skipgram. Baseline model with the mnemonic vectors initialized using the pretrained embeddings generated using the Skip-gram model. See Fig. 11.
- CNN-cbow. Baseline model with the mnemonic vectors initialized using the pretrained embeddings generated using the CBOW model. See Fig. 12.

According to the experiment, the model that achieved the highest cross-validation accuracy and macro F1-score is the one whose weights were randomly initialized from a uniform distribution (See Table 6). Therefore, to construct the multimodal network we decided not to initialize the embeddings with pretrained vectors. Nevertheless, all models achieved comparable results.

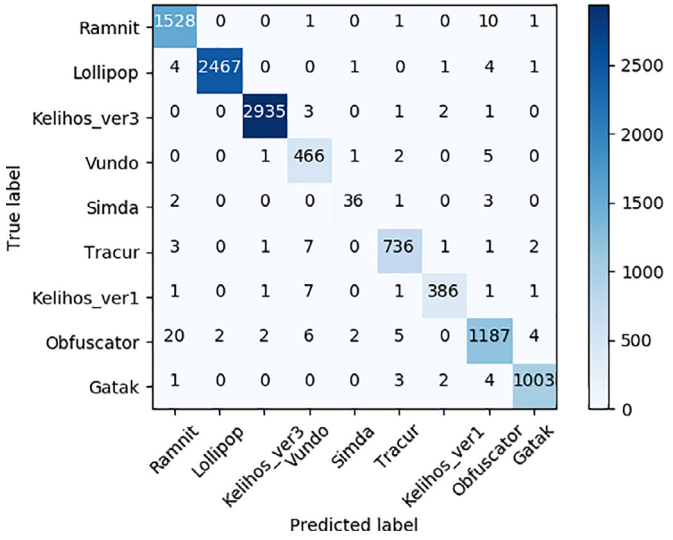

Fig. 12. CNN-cbow confusion matrix.

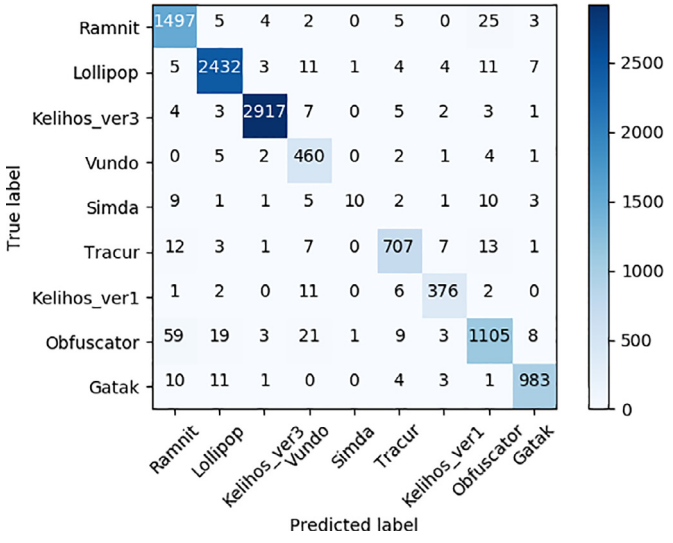

Fig. 13. MalConv confusion matrix.

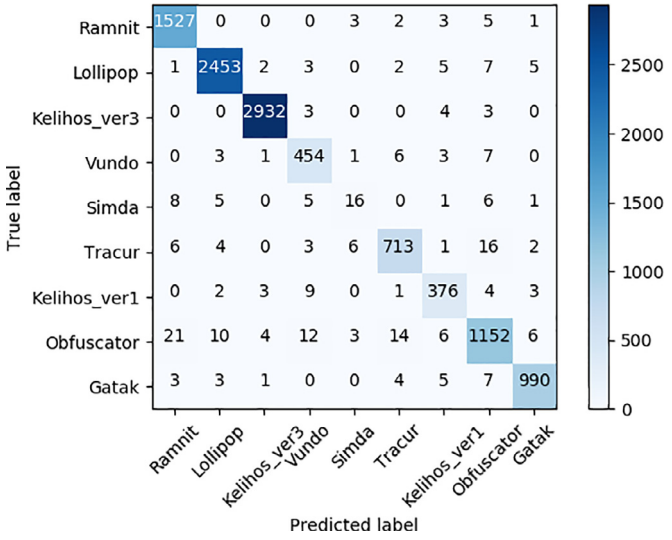

Fig. 14. DeepConv confusion matrix.

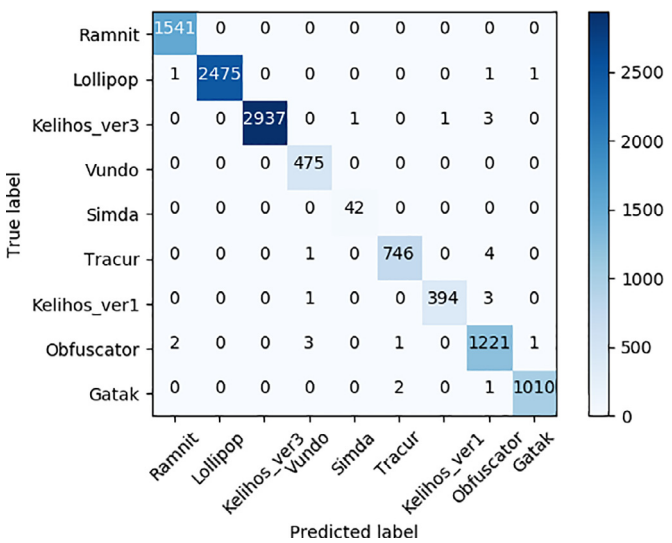

Fig. 15. HYDRA (pretraining, modality dropout) confusion matrix.

6.3. Byte-based component performance

In this section we compare the performance of the MalConv and DeepConv's model on the Microsoft Malware Classification Challenge dataset (Ronen et al., 2018). Figs. 13 and 14 display the confusion matrices reported from 10-fold cross-validation, whose macro F1-score is 0.8902 and 0.9071 for the MalConv and DeepConv model's, respectively (See Table 7). As a result, the byte-based component of our multimodal system would consist of the DeepConv architecture, given its superior performance.

The performance of these approaches is slightly worse than that of the approaches based on the assembly language source code. This is partially due to the high-dimensionality of the input sequence and the reduced training set, which make byte-based approaches suffer severely from overfitting. However, the hexadecimal representation of the malware's binary content is a very important source of information for any classifier as it is the minimal type of representation that can obtained from an executable. Depending on the obfuscation techniques employed by malware authors, the assembly language source code might not be retrieved correctly. Under these circumstances, the only information available is provided by the hexadecimal representation of the malware's binary content. Consequently, the information extracted from this modality of information is crucial to be able to correctly classify those malicious executables.

Table 6
Opcode-based CNN approaches comparison.

Approach	Accuracy	Macro F1-score
Random guess	0.1755	–
Zero rule	0.2707	–
CNN-rand	0.9917	0.9856
CNN-skipgram	0.9899	0.9770
CNN-cbow	0.9886	0.9717

Table 7
Bytes-based approaches comparison.

Approaches	Accuracy	Macro F1-score
Random guess	0.1755	–
Zero rule	0.2707	–
MalConv	0.9641	0.8902
DeepConv	0.9756	0.9071

Table 8
10-fold cross validation accuracy and macro F1-score comparison of modality-based and HYDRA models.

Model	Accuracy	Macro F1-score
API-based Feedforward Network	0.9833	0.9621
Assembly-based Shallow CNN	0.9917	0.9856
Bytes-based DeepConv	0.9756	0.8902
HYDRA	0.9871	0.9695
HYDRA (Pretraining, Modality Dropout)	**0.9975**	**0.9954**

6.4. Effectiveness of the multimodal deep learning model

In this section, we demonstrate the effectiveness of HYDRA by comparing it to the models trained on each modality independently. Fig. 15 shows the jointly reported accuracies of the 10-fold cross validation procedure to estimate the performance of HYDRA. The results of all models are shown in Table 8. We can observe that HYDRA's overall accuracy is 0.9975 and the macro F1-score is 0.9954, outperforming the modality-specific model's. Per-modality pretraining and multimodal dropout have been critical for the success of HYDRA. On the one hand, per-modality pretraining avoids overfitting one subset of features belonging to one modality and underfitting the features belonging to the others. On the other hand, multimodal dropout prevents the co-adaptation of the subnetworks to a specific feature type or modality of data.

6.4.1. Comparison with the state-of-the-art

To further evaluate the performance of our multimodal approach, we compared HYDRA with state-of-the-art methods in the literature that have evaluated their models on the dataset provided for the Kaggle's Microsoft Malware Classification Challenge. The results are shown in Table 9.

Methods in the literature are divided into various groups depending on the feature types used as input for the training algorithms. The groups are as follows:

- IMG-based approaches. This group consists of methods that take as input a grayscale image representing the malware's binary content (Gibert et al., 2018c) or a set of features extracted from it using any feature extractor technique (Ahmadi et al., 2016; Narayanan et al., 2016).
- Entropy-based approaches (Ahmadi et al., 2016; Gibert et al., 2018b) analyze the entropy and structural entropy representation of malware.
- Opcode-based approaches are split into two groups, (1) traditional approaches that extract n-gram features (Ahmadi et al., 2016) and (2) deep learning approaches (Gibert et al., 2017; 2019; McLaughlin et al., 2017), that take as input a sequence of

Table 9
10-fold cross validation accuracy of methods evaluated on the microsoft malware classification challenge.

Approach	Feature Type	Classification Algorithm	Accuracy	Macro F1
Random guess	–	–	0.1755	–
Zero rule	–	–	0.2707	–
Grayscale image				
Gibert et al. (2018c)	128 × 128 Grayscale Image	CNN	0.9750	0.9400
Ahmadi et al. (2016)	Haralick features	XGBoost	0.9690	0.9282
Ahmadi et al. (2016)	Local Binary Pattern features	XGBoost	0.9724	0.9530
Narayanan et al. (2016)	PCA features	1-NN	0.9660	0.9102
Entropy				
Gibert et al. (2018b)	Structural Entropy	Dynamic Time Warping +K-NN	0.9894	0.9813
Gibert et al. (2018b)	Structural Entropy	CNN	0.9828	0.9636
Ahmadi et al. (2016)	Entropy Statistical Measures	XGBoost	0.9900	0.9766
Sequence of opcodes				
Ahmadi et al. (2016)	1-Gram	XGBoost	0.9929	0.9906
McLaughlin et al. (2017)	Opcode sequence	CNN	0.9903	0.9743
Gibert et al. (2019)	Opcode sequence	Hierarchical CNN	0.9913	0.9830
OPCODE-based component	Opcode sequence	CNN	0.9917	0.9856
Sequence of bytes				
Ahmadi et al. (2016)	1-Gram	XGBoost	0.9850	0.9678
Raff et al. (2018)	Bytes sequence	MalConv	0.9641	0.8894
BYTE-based component (Krčál et al., 2018)	Bytes sequence	DeepConv	0.9756	0.9089
Le et al. (2018)*	Scaled bytes sequence	CNN	0.9647	0.9341
Le et al. (2018)*	Scaled bytes sequence	CNN+Unidirectional LSTM	0.9800	0.9577
Le et al. (2018)*	Scaled bytes sequence	CNN+Bidirectional LSTM	0.9814	0.9662
Gibert et al. (2018a)	Bytes sequence	Denoising Autoencoder + Dilated Residual Network	0.9861	0.9719
Yousefi-Azar et al. (2017)	1-Gram	Autoencoder + XGBoost	0.9309	0.8664
API invocations				
Ahmadi et al. (2016)	API feature vector (796)	XGBoost	0.9868	0.9638
API-based component	API feature vector (4500)	Feed-forward network	0.9833	0.9621
Multiple features				
Zhang et al. (2016)	Total lines of each Section, Operation Code Count, API Usage, Special Symbols Count, Asm File Pixel Intensity Feature, Bytes File Block Size Distribution, Bytes File N-Gram	Ensemble Learning (XGBoost)	0.9974	0.9938
Ahmadi et al. (2016)	ENT, Bytes 1-G, STR, IMG1, IMG2, MD1, MISC, OPC, SEC, REG, DP, API, SYM, MD2	Ensemble Learning (XGBoost)	0.9976	0.9931
Mays et al. (2017)	IMG and Opcode N-Grams	Ensemble Learning (CNN and NN)	0.9724	0.9618
HYDRA	**APIs, Bytes sequence, Opcode sequence**	**Multimodal Deep Neural Network**	**0.9975**	**0.9951**

opcodes representing the malware's assembly language source code.
- Byte-based approaches are split similarly to opcode-based approaches. On the one hand, there are those approaches that extract n-gram features from the bytes sequence (Ahmadi et al., 2016). On the other hand, deep learning approaches jointly learn to extract features and classify malware during training (Krčál et al., 2018; Raff et al., 2018). Furthermore, it includes methods that learn an encoded representation of malware using autoencoders (Gibert et al., 2018a; Yousefi-Azar et al., 2017).
- API-based approaches (Ahmadi et al., 2016) generate a feature set by mining the API calls that a classifier uses to make predictions.
- Multimodal learning refers to those approaches that learn to detect malware using information from multiple modalities (Ahmadi et al., 2016; Mays et al., 2017; Zhang et al., 2016).

Furthermore, note that the most used classification algorithms are either neural networks or gradient boosting. On the one hand, neural networks and in particular convolutional neural networks have recently attracted the academic community due to their advantages on processing raw data and their ability to learn features by themselves. On the other hand, gradient boosting and, in particular, the XGBoost library have until recently provided unmatched performance in tasks that rely on feature engineering and domain-specific knowledge. This trend might change in the near future due to the availability of bigger training feeds for the research community and developments and improvements in the multimodal learning field. As observed in Table 9, HYDRA achieved comparable results to (Zhang et al., 2016) and (Ahmadi et al., 2016) with fewer input modalities and achieved a higher detection rate and macro F1-score than (Mays et al., 2017) and the remaining deep learning and feature engineering based approaches in the literature. Thus, we demonstrate that end-to-end learning systems can be successfully complemented with hand-engineered features to achieve state-of-the-art results in the malware classification task, where domain-specific knowledge has been the way-to-go for building systems.

7. Conclusions

In this paper, we present a novel malware classification framework that combines both hand-engineered features and end-to-end components in a modular architecture. To the best of our knowledge, this research is the first application of multimodal deep learning for malware classification, and in particular Portable Executable files. The multimodal approach learns and combines characteristics of malware from various sources of information, yielding to higher classification performance than those classifiers that take as input only a single modality of data. Three kinds of modalities are extracted by analyzing the hexadecimal representation of malware's binary content and its disassembly counterpart, (1) the list of API functions invoked, (2) the sequence of mnemonics representing malware's assembly language source code, and (3) the sequence of bytes representing malware's binary content. This architecture can be enriched with many more feature types to express executables' characteristics and it serves as a baseline model for future improvements. Furthermore, by fusing hand-crafted features with the end-to-end components we are able to mix the strengths of both approaches, characterizing domain-specific features and the ability of deep learning to automatically extract a set of descriptive features without relying on domains' knowledge.

Reported results allow the effectiveness of our approach to be assessed with respect to state-of-the-art techniques. The detection accuracy and macro f1-score of our multimodal deep learning architecture is comparable to gradient boosting methods based on feature engineering, with ours only relying on three basic types of features from Portable Executables, and far more accurate than deep learning approaches in the literature.

7.1. The problem of concept drift

Machine learning techniques were originally designed for stationary environments in which the training and test sets are assumed to be generated from the same statistical distribution. In a stationary environment, a model will approximate a mapping function $f(x)$ given input data x to predict an output value y, $y = f(x)$, and it is assumed that the mapping learned from the data will be valid in the future and the relationship between input and output do not change over time. However, this assumption is not valid in the malware domain. Software applications, including malware, evolve over time due to changes resulting from adding features and capabilities, fixing bugs, porting to new platforms, etc. Additionally, versions of the same software are expected to be similar to previous versions with few exceptions. Thus, the similarity between previous and future versions will degrade slowly over time. This is known as the problem of concept drift. Concept drift is the problem of the changing underlying relationships in the data. This will result in the decay of the prediction quality of malware detectors and classifiers over time as malware evolves and new variants appear (Pendlebury et al., 2019).

Furthermore, malware is constantly pushed to evolve in order to avoid detection by anti-malware engines and be able to infect new hosts. Thus, malware authors are well-motivated to intentionally craft adverarial examples (Huang et al., 2011) using a wide range of obfuscation techniques (You and Yim, 2010). As a result, the aforementioned issues have to be taken into account in the process of building a sustainable model for malware detection and classification (Jordaney et al., 2017).

7.2. Future work

One future line of research could be the implementation of new architectures to detect and classify malware from their binary content represented as a sequence of bytes, as current methods perform below average in comparison with the rest of approaches in the literature. A second line of research could be to study the incorporation of more data modalities or feature types into the current multimodal architecture and analyze its impact on the performance of the system. A third line of research could be the study of explainable artificial intelligence (XAI) techniques to interpret the results of machine learning models for malware detection and classification in order to help security researchers in the process of malware analysis.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Daniel Gibert: Conceptualization, Software, Validation, Formal analysis, Investigation, Data curation, Writing - original draft, Writing - review & editing. **Carles Mateu:** Supervision, Writing - original draft, Writing - review & editing, Resources, Funding acquisition. **Jordi Planes:** Supervision, Writing - original draft, Writing - review & editing, Resources, Funding acquisition.

Acknowledgments

This research has been partially funded by the Spanish MICINN Projects TIN2015-71799-C2-2-P, ENE2015-64117-C5-1-R, and is supported by the University of Lleida. This research article has received a grant (2019 call) from the University of Lleida Language Institute to review the English.

References

Aafer, Y., Du, W., Yin, H., 2013. DroidAPIMiner: mining API-level features for robust malware detection in android. In: Zia, T., Zomaya, A., Varadharajan, V., Mao, M. (Eds.), Security and Privacy in Communication Networks. Springer International Publishing, Cham, pp. 86–103.

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X., 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Ahmadi, M., Ulyanov, D., Semenov, S., Trofimov, M., Giacinto, G., 2016. Novel feature extraction, selection and fusion for effective malware family classification. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. ACM, New York, NY, USA, pp. 183–194. doi:10.1145/2857705.2857713.

Bidoki, S.M., Jalili, S., Tajoddin, A., 2017. PbMMD: a novel policy based multi-process malware detection. Eng. Appl. Artif. Intell. 60, 57–70. doi:10.1016/j.engappai.2016.12.008.

Chandrasekar, K., Cleary, G., Cox, O., Lau, H., Nahorney, B., Gorman, B.O., O'Brien, D., Wallace, S., Wood, P., Wueest, C., 2017. Internet Security Threat Report. Technical Report. Symantec Corporation.

Cleary, G., Corpin, M., Cox, O., Lau, H., Nahorney, B., O'Brien, D., O'Gorman, B., Power, J.-P., Wallace, S., Wood, P., Wueest, C., 2018. Internet Security Threat Report. Technical Report. Symantec Corporation.

Clevert, D., Unterthiner, T., Hochreiter, S., 2015. Fast and accurate deep network learning by exponential linear units (ELUS). CoRR arXiv:1511.07289

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P., 2011. Natural language processing (almost) from scratch. J. Mach. Learn. Res. 12, 2493–2537.

Daniely, Y., Clayton, R., Johnson, S., Eisner, T., Fishman, Y., Kaye, J., 2018. Security Report. Technical Report. Check Point Software Technologies Ltd.

Ghiasi, M., Sami, A., Salehi, Z., 2015. Dynamic VSA: a framework for malware detection based on register contents. Eng. Appl. Artif. Intell. 44, 111–122. doi:10.1016/j.engappai.2015.05.008.

Gibert, D., Bejar, J., Mateu, C., Planes, J., Solis, D., Vicens, R., 2017. Convolutional neural networks for classification of malware assembly code. In: International Conference of the Catalan Association for Artificial Intelligence, pp. 221–226. doi:10.3233/978-1-61499-806-8-221.

Gibert, D., Mateu, C., Planes, J., 2018. An end-to-end deep learning architecture for classification of malware's binary content. In: Artificial Neural Networks and Machine Learning - ICANN 2018 - 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4–7, 2018, Proceedings, Part III, pp. 383–391. doi:10.1007/978-3-030-01424-7_38.

Gibert, D., Mateu, C., Planes, J., 2019. A hierarchical convolutional neural network for malware classification. In: The International Joint Conference on Neural Networks 2019. IEEE, pp. 1–8.

Gibert, D., Mateu, C., Planes, J., Vicens, R., 2018. Classification of malware by using structural entropy on convolutional neural networks. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2–7, 2018, pp. 7759–7764.

Gibert, D., Mateu, C., Planes, J., Vicens, R., 2018. Using convolutional neural networks for classification of malware represented as images. J. Comput. Virol. Hacking Tech. doi:10.1007/s11416-018-0323-0.

Glorot, X., Bengio, Y., 2010. Understanding the difficulty of training deep feedforward neural networks. In: In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics.

Hassen, M., Carvalho, M.M., Chan, P.K., 2017. Malware classification using static analysis based features. In: 2017 IEEE Symposium Series on Computational Intelligence (SSCI), pp. 1–7. doi:10.1109/SSCI.2017.8285426.

Hassen, M., Chan, P.K., 2017. Scalable function call graph-based malware classification. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy. ACM, New York, NY, USA, pp. 239–248. doi:10.1145/3029806.3029824.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., 2012. Improving neural networks by preventing co-adaptation of feature detectors. CoRR arXiv:1207.0580

Hu, X., Jang, J., Wang, T., Ashraf, Z., Stoecklin, M.P., Kirat, D., 2016. Scalable malware classification with multifaceted content features and threat intelligence. IBM J. Res. Dev. 60 (4), 6:1–6:11. doi:10.1147/JRD.2016.2559378.

Huang, L., Joseph, A.D., Nelson, B., Rubinstein, B.I., Tygar, J.D., 2011. Adversarial machine learning. In: Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence. Association for Computing Machinery, New York, NY, USA, pp. 43–58. doi:10.1145/2046684.2046692.

Jain, S., Meena, Y.K., 2011. Byte level n–gram analysis for malware detection. In: Venugopal, K.R., Patnaik, L.M. (Eds.), Computer Networks and Intelligent Computing. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 51–59.

Jordaney, R., Sharad, K., Dash, S.K., Wang, Z., Papini, D., Nouretdinov, I., Cavallaro, L., 2017. Transcend: detecting concept drift in malware classification models. In: 26th USENIX Security Symposium (USENIX Security 17). USENIX Association, Vancouver, BC, pp. 625–642.

Khan, R.U., Zhang, X., Kumar, R., 2018. Analysis of ResNet and GoogleNet models for malware detection. J. Comput. Virol. Hacking Tech. doi:10.1007/s11416-018-0324-z.

Kinable, J., Kostakis, O., 2011. Malware classification based on call graph clustering. J. Comput. Virol. 7 (4), 233–245. doi:10.1007/s11416-011-0151-y.

Klambauer, G., Unterthiner, T., Mayr, A., Hochreiter, S., 2017. Self-normalizing neural networks. CoRR arXiv:1706.02515

Krčál, M., Švec, O., Bálek, M., Jašek, O., 2018. Deep convolutional malware classifiers can learn from raw executables and labels only.

Le, Q., Boydell, O., Namee, B.M., Scanlon, M., 2018. Deep learning at the shallow end: malware classification for non-domain experts. Digit. Invest. 26, S118–S126. doi:10.1016/j.diin.2018.04.024.

Lyda, R., Hamrock, J., 2007. Using entropy analysis to find encrypted and packed malware. IEEE Secur. Priv. 5 (2), 40–45. doi:10.1109/MSP.2007.48.

Mays, M., Drabinsky, N., Brandle, S., 2017. Feature selection for malware classification. In: Proceedings of the 28th Modern Artificial Intelligence and Cognitive Science Conference 2017, Fort Wayne, IN, USA, April 28–29, 2017., pp. 165–170.

McLaughlin, N., Martinez del Rincon, J., Kang, B., Yerima, S., Miller, P., Sezer, S., Safaei, Y., Trickel, E., Zhao, Z., Doupé, A., Joon Ahn, G., 2017. Deep android malware detection. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy. ACM, New York, NY, USA, pp. 301–308. doi:10.1145/3029806.3029823.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J., 2013. Distributed representations of words and phrases and their compositionality. In: Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2. Curran Associates Inc., USA, pp. 3111–3119.

Moskovitch, R., Stopel, D., Feher, C., Nissim, N., Elovici, Y., 2008. Unknown malcode detection via text categorization and the imbalance problem. In: 2008 IEEE International Conference on Intelligence and Security Informatics, pp. 156–161. doi:10.1109/ISI.2008.4565046.

Narayanan, B.N., Djaneye-Boundjou, O., Kebede, T.M., 2016. Performance analysis of machine learning and pattern recognition algorithms for malware classification. In: 2016 IEEE National Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS), pp. 338–342. doi:10.1109/NAECON.2016.7856826.

Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B.S., 2011. Malware images: Visualization and automatic classification. In: Proceedings of the 8th International Symposium on Visualization for Cyber Security. ACM, New York, NY, USA, pp. 4:1–4:7. doi:10.1145/2016904.2016908.

Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L., 2019. TESSERACT: eliminating experimental bias in malware classification across space and time. In: 28th USENIX Security Symposium (USENIX Security 19). USENIX Association, Santa Clara, CA, pp. 729–746.

Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., Nicholas, C.K., 2018. Malware detection by eating a whole EXE. In: The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2–7, 2018., pp. 268–276.

Ronen, R., Radu, M., Feuerstein, C., Yom-Tov, E., Ahmadi, M., 2018. Microsoft malware classification challenge. CoRR arXiv:1802.10135

Salehi, Z., Sami, A., Ghiasi, M., 2017. Maar: robust features to detect malicious activity based on API calls, their arguments and return values. Eng. Appl. Artif. Intell. 59, 93–102. doi:10.1016/j.engappai.2016.12.016.

Sami, A., Yadegari, B., Rahimi, H., Peiravian, N., Hashemi, S., Hamze, A., 2010. Malware detection based on mining API calls. In: Proceedings of the 2010 ACM Symposium on Applied Computing. ACM, New York, NY, USA, pp. 1020–1025. doi:10.1145/1774088.1774303.

Santos, I., Brezo, F., Ugarte-Pedrero, X., Bringas, P.G., 2013. Opcode sequences as representation of executables for data-mining-based unknown malware detection. Information Sciences 231, 64–82. doi:10.1016/j.ins.2011.08.020. Data Mining for Information Security

Shabtai, A., Moskovitch, R., Feher, C., Dolev, S., Elovici, Y., 2012. Detecting unknown malicious code by applying classification techniques on opcode patterns. Secur. Inform. 1 (1), 1. doi:10.1186/2190-8532-1-1.

Sorokin, I., 2011. Comparing files using structural entropy. J. Comput. Virol. 7 (4), 259. doi:10.1007/s11416-011-0153-9.

Souri, A., Hosseini, R., 2018. A state-of-the-art survey of malware detection approaches using data mining techniques. Hum. Centric Comput. Inf. Sci. 8 (1), 3. doi:10.1186/s13673-018-0125-x.

Ucci, D., Aniello, L., Baldoni, R., 2019. Survey of machine learning techniques for malware analysis. Comput. Secur. 81, 123–147. doi:10.1016/j.cose.2018.11.001.

Vulnerabilities, C., Exposures, 2016. CVE-2017-0143.Available from MITRE, CVE-ID CVE-2017-0143.

Wojnowicz, M., Chisholm, G., Wolff, M., Zhao, X., 2016. Wavelet decomposition of software entropy reveals symptoms of malicious code. J. Innov. Digit. Ecosyst. 3 (2), 130–140. doi:10.1016/j.jides.2016.10.009.

You, I., Yim, K., 2010. Malware obfuscation techniques: a brief survey. In: 2010 International Conference on Broadband, Wireless Computing, Communication and Applications, pp. 297–300. doi:10.1109/BWCCA.2010.85.
Yousefi-Azar, M., Varadharajan, V., Hamey, L., Tupakula, U., 2017. Autoencoder-based feature learning for cyber security applications. In: 2017 International Joint Conference on Neural Networks (IJCNN), pp. 3854–3861. doi:10.1109/IJCNN.2017.7966342.
Yuxin, D., Siyi, Z., 2017. Malware detection based on deep learning algorithm. Neural Comput. Appl. doi:10.1007/s00521-017-3077-6.
Zhang, Y., Huang, Q., Ma, X., Yang, Z., Jiang, J., 2016. Using multi-features and ensemble learning method for imbalanced malware classification. In: 2016 IEEE Trustcom/BigDataSE/ISPA, pp. 965–973. doi:10.1109/TrustCom.2016.0163.

Daniel Gibert is a Ph.D. student at the Department of Computer Science and Industrial Engineering of the University of Lleida, in Spain. He graduated in 2014 and received a Masters degree in Artificial Intelligence in 2016 from the Polytechnic University of Catalonia. His research interests include intrusion detection and machine learning.

Carles Mateu received his B.Sc. from Universitat de Lleida, M.Sc. from Open University of Catalonia, and Ph.D. from University of Lleida in 2009. He is currently an Associate Professor at the University of Lleida. His research interests include Security, Energy Storage, Energy Efficiency and Artificial Intelligence.

Jordi Planes received his B.Sc. from Universitat de Lleida, M.Sc. from University Rovira i Virgili, and Ph.D. from University of Lleida in 2007. He is currently an Associate Professor at the University of Lleida. His research interests include Applications of Neural Networks.