

Universitat de Lleida

Document downloaded from:

<http://hdl.handle.net/10459.1/71360>

The final publication is available at:

<https://doi.org/10.1016/j.cose.2020.102159>

Copyright

(c) Elsevier, 2020

Auditing Static Machine Learning Anti-Malware Tools Against Metamorphic Attacks

Daniel Gibert^{a,*}, Carles Mateu^a, Jordi Planes^a, Joao Marques-Silva^b

^a*University of Lleida, Jaume II, 69, Lleida, Spain*

^b*Université Fédérale Toulouse Midi-Pyrénées, 41 Allées Jules Guesde, Toulouse, France*

Abstract

Malicious software is one of the most serious cyber threats on the Internet today. Traditional malware detection has proven unable to keep pace with the sheer number of malware because of their growing complexity, new attacks and variants. Most malware implement various metamorphic techniques in order to disguise themselves, therefore preventing successful analysis and thwarting the detection by signature-based anti-malware engines. During the past decade, there has been an increase in the research and deployment of anti-malware engines powered by machine learning, and in particular deep learning, due to their ability to handle huge volumes of malware and generalize to never-before-seen samples. However, there is little research about the vulnerability of these models to adversarial examples. To fill this gap, this paper presents an exhaustive evaluation of the state-of-the-art approaches for malware classification against common metamorphic attacks. Given the limitations found in deep learning approaches, we present a simple architecture that increases 14.95% the classification performance with respect to MalConv's architecture. Furthermore, the use of the metamorphic techniques to augment the training set is investigated and results show that it significantly improves the classification of malware belonging to families with few samples.

Keywords: Malware Analysis, Malware Classification, Software Obfuscation, N-gram Extraction, Machine Learning, Deep Learning

1. Introduction

In today's ever-connected society, cyber-attacks have been dramatically increasing

in number and damage up to the point that cyberthreats are ranked as a consistent and persistent threat among the top global risks¹, along with weather extremes, climate change and natural disasters. Some estimates² predict that the cost of cyber-crime to the world would be 6 trillion an-

*I am corresponding author

Email addresses:

daniel.gibert@diei.udl.cat (Daniel Gibert),
carlesm@diei.udl.cat (Carles Mateu),
jplanes@diei.udl.cat (Jordi Planes),
joao.marques-silva@univ-toulouse.fr (Joao Marques-Silva)

¹http://www3.weforum.org/docs/WEF_Global_Risks_Report_2019.pdf

²<https://cybersecurityventures.com/>

nually by 2021, rising from the 3 trillion in 2015. This estimate includes damage and destruction of data, stolen money, lost productivity, theft of personal, financial data and intellectual property, fraud, disruption of the normal course of business, forensic investigation and reputational harm. According to MalwareBytes³, there has been an increase of 13% in the business threat detections in 2019 with a dramatic spike in detections of the malware Emotet at the beginning of the year.

Malicious software is one of the most common methods employed by cybercriminals to launch a cyberattack. Other methods include, but are not limited to, phishing, man-in-the-middle attack, SQL injection, etc. Every day, the AV-TEST Institute registers over 350000 new malicious programs and potentially unwanted applications (PUA). In fact, the number of total malware has more than doubled from the 470.01m in 2015 to 1065.61m in 2020⁴. However, this is not due to an increase in new malware but to the reuse of well-established families through the usage of code obfuscation techniques to bypass detection engines. Thus, to keep up with malware and be able to reduce its impact, it is necessary to improve the computer systems' cyberdefenses and in particular, anti-malware engines, the last layer of defense against a cyberattack and the defensive layer responsible of preventing, detecting and removing malicious software.

The fastest and most reliable method employed by anti-malware engines to detect

known malware is by means of unique signatures. Signatures are composed by sequences of bytes or data, to provide an identifier for each malicious software or group of samples with similar capabilities or behavior. However, signatures cannot detect against unknown malware because a new signature has to be developed previously. Consequently, signature-based detection only protects against known malware. Another problem is that malware can alter its signature to avoid detection by simply modifying the code while preserving its functionality and behavior. Furthermore, as new malware appears every day, it is necessary to store large amounts of signatures, demanding considerable storage, making slow to search a particular signature, and affecting system performance (Amro and Alkhalfah, 2015).

Due to the sheer volumes of new malware variants being deployed every day, anti-malware solutions that rely solely on signatures have become obsolete. During the past decade, there has been an increase in the research and deployment of anti-malware engines powered by machine learning (Gibert et al., 2020b; Ucci et al., 2019; Souri and Hosseini, 2018) to complement signature-based detection due to their ability to handle huge volumes of data and generalize to never-before-seen malware. This is achieved by summarizing complex relationships among features that are discriminative between malware and goodware or between malware families, allowing the detection engine to adapt to the modifications in malware's code. However, there is little research about the susceptibility of these models to adversarial samples. Most state-of-the-art approaches (Demetrio et al., 2019; Kolosnjaji et al., 2018; Suciu et al., 2019) investigated how slight permutations

³https://resources.malwarebytes.com/files/2020/02/2020_State-of-Malware-Report.pdf

⁴<https://www.av-test.org/en/statistics/malware/>

generated by appending bytes at the end of sections or at the end of the file can produce misclassifications. Unfortunately, these approaches are based on modifications that only affect the structure of the Portable Executable files. None of them modify the actual source code of the executables. Thus, they greatly differ from the modifications performed by real-word malware to generate variants of itself.

The aim of this paper is to fill this gap. To this end, this work provides an extensive evaluation of state-of-the-art detectors powered by machine learning (ML) against common metamorphic techniques. More specifically, the performance of the ML approaches is assessed against the modifications performed by the following metamorphic techniques:

- The dead code insertion technique.
- The registers reassignment technique.
- The subroutine reordering technique.
- The code reordering through jumps technique.

Given the poor performance of the byte-based deep learning approaches in the literature (Raff et al., 2018a; Krčál et al., 2018; Gibert et al., 2018) in comparison to the opcode-based approaches (Gibert et al., 2017) due to their greater complexity and the length of the input data, we propose a shallow architecture that improves 14.95% and 16.38% with respect to MalConv (Raff et al., 2018a) and DeepConv (Krčál et al., 2018) architectures. Instead of learning complex and deep patterns, our architecture learns n-gram like features from the malware’s binary content represented as a sequence of bytes. This is achieved through a convolutional layer with filters of various

sizes that act as feature extractors. Furthermore, we investigate the usage of the aforementioned metamorphic techniques to augment the dataset and reduce class imbalance. The generalization performance of the ML approaches has been evaluated on a standard public benchmark provided by Microsoft for the Big Data Innovators Gathering Anti-Malware Prediction Challenge (Ronen et al., 2018) for reproducibility purposes.

The rest of the paper is organized as follows. Section 2 provides the background. Section 3 introduces state-of-the-art approaches to bypass ML malware detectors. Section 4 describes the metamorphic techniques employed by malware authors to modify the executables. Section 5 presents the ML approaches evaluated in this work. Section 6 presents the results of the experimentation. Finally, Section 7 summarizes the concluding remarks and presents some future lines of research.

2. Background

This section introduces the task of malware classification, it presents an overview of malware, the Portable Executable (PE) file format and the methods employed by malware authors to evolve malware.

2.1. The Task of Malware Detection and Classification

Malware detection refers to the task of identifying whether or not a given file is malicious to a computer system. By malicious we refer to code that is harmful to the system. Malware might seek to invade, damage or disable partially or completely the computer system, often taking control of it. Depending on their purpose, malware can be divided into various, not mutually exclusive

general categories including, but not limited to adware, spyware, trojans, rootkit, virus, worms, ransomware, etc. These categories provide a general overview of the functionality and behavior of malware. Typically, malware is also identified by a family name. A malware family refers to a collection of malware that has been generated from the same code base. Furthermore, malware families are divided into variants or strains, that is, malware built from an existing code base that have different signatures that are not included in the list of signatures used by anti-malware solutions. Distinguishing and classifying different types of malware is known as the task of malware classification and it provides information to better understand how the malware has infected the computers or devices, their threat level and how to protect against them.

2.2. The Portable Executable File Format

Malicious software targeting the Windows operating system is commonly written using the Portable Executable (PE) format, a file format for executable, object code, DLLs and others used in 32-bit and 64-bit versions of the Windows operations system. Portable Executables contain the information necessary for the Windows operating system to run the executable code including dynamic library references for linking, API export and import tables, etc. A Portable Executable file consists of headers and sections that tell the dynamic linker how to map the file into memory. An overview of the PE file is shown in Figure 1. The PE Header includes information regarding to the number of sections, their sizes, characteristics of the file, the import address table (IAT), etc. Furthermore, the PE file is divided into sections that contain the code

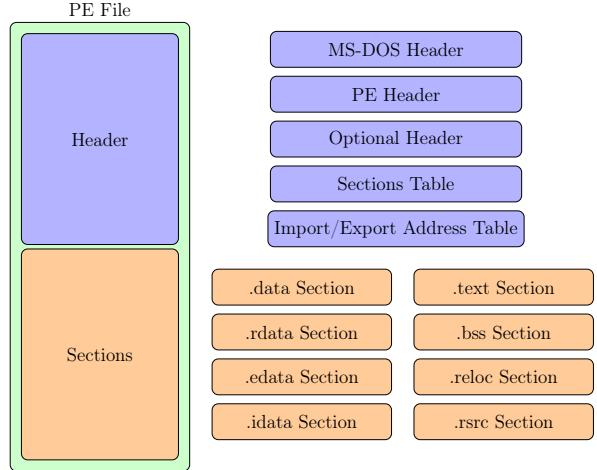


Figure 1: Portable Executable File Format

and data of the executable, including, but not limited to:

- the .text section. This section keeps the actual code of the computer program although the code can be written in any other section.
- the .data section. This section is used to declare initialized data or constants that do not change at runtime.
- the .rsrc section. This section contains all the resources of the program.

Detailed information of the PE file format can be found in the documentation provided by Microsoft⁵.

2.3. Malware Evolution

Malware is constantly evolving and seeking new ways to bypass detection engines. The proliferation of malware has increased mainly due to the use of polymorphic and metamorphic techniques employed by malware authors to evade detection and hide

⁵<https://docs.microsoft.com/en-us/windows/win32/debug/pe-format>

the true behavior of the executables. In polymorphic malware, a polymorphic engine is used to mutate the code while keeping the original functionality intact. The two most common methods to hide code are packing and encryption. On the one hand, packers employ one or more layers of compression to hide the real code of the program. Then, the original code is restored into memory at runtime through the unpacking routines and it is executed. On the other hand, crypters encrypt malware or part of its code to make it harder to analyze. A crypter typically contains a stub used to encrypt and decrypt malicious code. On the contrary, metamorphic malware rewrites its code to an equivalent version each time it is propagated. Malware authors may employ various transformation techniques including, but not limited to, register renaming, subroutine reordering and garbage code insertion. Thanks to the combination of the aforementioned techniques, malware volumes rapidly grown, making forensic investigations of malware cases time-consuming, costly and difficult even for security analysts and experts.

The aforementioned circumstances force security analysts and researchers to continually improve their cyberdefenses to keep pace with the evolution of malware. This caused the following problems with traditional antivirus solutions that relied on signature-based and heuristic/behavioral methods. First, signatures cannot be used to detect unknown malware variants because a new signature has to be developed previously. Second, although behavior-based detection is an effective approach to analyze the file’s characteristics and behavior to determine if the file is indeed malware, the scanning and analysis is very time-consuming and can’t be applied to ev-

ery suspicious sample. Thus, researchers started adopting machine learning to complement their solutions and overcome the prior pitfalls of traditional signature-based engines and to provide an initial screening of the samples that exhibit malicious traits, as machine learning is well suited for processing large volumes of data.

3. Related Work

Machine learning has become an appealing tool for anti-malware vendors for either primary detection engines or as complementary detection heuristics. This is due to the ability of machine learning models to generalize to new samples, if the models are properly regularized. Furthermore, machine learning models allow to automatically summarize complex relationships among features that are discriminative between malware and goodware or between malware families, depending on the task, which allows the detection engine to adapt to the modifications in the malicious samples. For a complete review of machine learning solutions to detect and classify malware the reader is referred to the following articles (Souri and Hosseini, 2018; Ucci et al., 2019; Gibert et al., 2020b).

Although over the past decade there has been an increase in the research and deployment of machine learning solutions to tackle the problem of malware detection and classification, there is little research about the vulnerability of these models to adversarial attacks (Pitropakis et al., 2019; McGraw et al., 2019). Most state-of-the-art approaches (Demetrio et al., 2019; Kolosnjaji et al., 2018; Suciu et al., 2019) investigated how to slightly perturbate Portable Executable files by appending carefully-selected bytes at the end of the PE header

or at the end of the file, to evade detection by a shallow CNN architecture based on the raw bytes of the executable (Raff et al., 2018b). For instance, Demetrio et al. (2019) perturbed the bytes in the PE header that are expected to maximally increase the probability of evasion. On the other hand, Kolosnjaji et al. (2018) appended carefully-selected bytes at the end of the file, where these bytes were selected in a way that the resulting file minimizes the confidence associated to the malicious class. Unfortunately, for both approaches to work they need to have access to the machine learning model’s gradients, which is very unlikely, if not impossible, to happen in a real-world scenario. Furthermore, some approaches in the literature tried to automatically learn which changes to perform to a feature vector (Hu and Tan, 2017) or to the actual executable (Anderson et al., 2018) in order to bypass black-box detectors. More specifically, Hu and Tan (2017) proposed a GAN to generate adversarial examples by modifying a binary feature vector, whose features refer to the API functions added to the import address table of the PE header of an executable. For example, if M APIs are used as features, an M -dimensional feature vector is constructed, with all the features corresponding to the imported API functions set to 1, and the rest set to 0. Contrarily, Anderson et al. (2018) proposed a reinforcement learning agent equipped with a set of functionality preserving operations like adding a function to the import address, manipulate the names of the sections, append bytes at the end of the file or between sections, etc. However, these modifications only affect the structure of the Portable Executables. None of them modify the actual source code of the executables. Thus, they greatly differ from the modifications per-

formed by real-world malware to generate new variants of themselves.

The address the limitations of the aforementioned adversarial attacks, this paper performs an extensive investigation of the vulnerability of state-of-the-art machine learning approaches to realistic attacks already being employed by malware authors to bypass detection. For a complete description of the attacks see Section 4.

4. Metamorphic Attacks

Malware authors usually employ metamorphic and polymorphic techniques to change the form of each instance from generation to generation in order to evade signature-based and pattern-matching detection. On the one hand, polymorphic malware pairs with a polymorphic engine with self-propagating code to continually change its appearance by using encryption or packaging algorithms to hide its code. On the other hand, metamorphic malware rewrites its code so that the newly propagated version of itself no longer matches its previous iterations. Metamorphic malware may use multiple transformation techniques that include, but are not limited to, garbage code or dead code insertion, register reassignment, subroutine reordering and code reordering through jumps. In this work we focus on the transformations performed by metamorphic techniques because they are the ones that make alterations to the actual source code of malware. Following, the most common metamorphic techniques are described in more detail.

4.1. Dead Code Insertion

The insertion of dead code or do-nothing instructions change the appearance of a program while not affecting the execution

of the original code. Examples of malware that added dead code instructions on each generation are Evol and MetaPHOR. Cf. Figure 2. Following are described the dead code instructions implemented for our research purposes.

- **NOP.** The NOP or no-op instruction (short for **no operation**) is an assembly language instruction that does nothing.
- **MOV Reg, Reg.** The MOV instruction copies the contents of the register referred by its second operand into the register referred to by its first operand.
- **PUSH Reg; POP Reg.** The PUSH instruction places the register referred to by its operand onto the top of the stack in memory while the POP instruction removes the element from the top of the stack.
- **ADD Reg, 0.** The ADD instruction adds the first and second operands, storing the result in its first operand.
- **SUB Reg, 0.** The SUB instruction subtracts the second operand from the first operand and stores the result in its first operand.
- **INC Reg; DEC Reg.** The INC instruction increments the content of the register referred by its operand by one while the DEC instruction decrements the contents of the register by one.
- **SHL Reg, 0.** The SHL instruction shifts the bits in its first operand's contents left, padding the resulting empty bit positions with zero. The number of bits to shift is specified by the second operand.

- **SHR Reg, 0.** The SHR instruction shifts the bits in its first operand's contents right, padding the resulting empty bit positions with zero. The number of bits to shift is specified by the second operand.

Notice that the dead code instructions **PUSH Reg; POP Reg** and **INC Reg; DEC Reg** do not necessarily need to be executed sequentially (one after the other). The instructions can be alternated with other instructions from the original code that do not modify the same registers as the do-nothing instructions.

```
.text:00470051 8B EC      mov    ebp, esp
.text:00470053 83 C4 98   add    esp, 0FFFFFF98h
.text:00470056 33 C0      xor    eax, eax
.text:00470058 8B 15 7C 10 4B 00  mov    edx, dword_4B107C
.text:0047005E 90          nop
.text:0047005F 89 55 EC   mov    [ebp+var_14], edx
.text:00470062 89 45 EC   mov    [ebp+var_14], eax
.text:00470065 53          push   ebx
.text:00470066 8B 1D 7D 10 4B 00  mov    ebx, dword_4B107C
.text:0047006C 83 FB 2D   cmp    ebx, 2Dh
.text:0047006F 75 03      jnz    short loc_470073
.text:00470071 89 5D EC   mov    [ebp+var_14], ebx
```

Figure 2: Assembly language source code after the insertion of a NOP instruction.

4.2. Register's Reassignment

Register reassignment switches registers from generation to generation while keeping the program behavior unaltered. Cf. Figure 3. This technique was first used by Win95/RegSwap virus. Traditional anti-virus engines detect viruses that employ this technique by a wildcard string search (Ször and Ferrie, 2001). Wildcard strings allow to skip particular bytes in regular expressions. For instance, in "89 ?? 7C 10 4B 00", the wildcard is indicated by '??'.

4.3. Subroutine Reordering

The subroutine reordering technique employs permutations to reorder the subroutines of a malware executable. Cf. Figure 4. With n different subroutines, this

```

.text:00470051 8B EC      mov ebp esp
.text:00470053 83 C4 98   add esp 0FFFFFF98h
.text:00470056 31 C9      xor ecx ecx
.text:00470058 8B 1D 7C 10 4B 00  mov ebx dword_4B107C
.text:0047005E 89 5D 10    mov [ebp+10h] ebx
.text:00470061 89 4D 10    mov [ebp+10h] ecx

.text:00470051 8B EC      mov ebp esp
.text:00470053 83 C4 98   add esp 0FFFFFF98h
.text:00470056 33 C0      xor eax eax
.text:00470058 8B 15 7C 10 4B 00  mov edx, dword_4B107C
.text:0047005E 89 55 EC    mov [ebp+10h], edx
.text:00470061 89 45 EC    mov [ebp+10h], eax

```

Figure 3: Register reassignment example. Registers ecx and ebx are switched to eax and edx, respectively.

technique can generate up to $n!$ different variants. The technique was initially employed by Win32/Ghost virus, which had ten subroutines. Thus, it could generate $10! = 3,628,800$ variants.

```

.text:00409333  loc_409333:
.text:00409333 85 C9      test    ecx, ecx
.text:00409335 74 1E      jz     short loc_409355
.text:00409337 8D 4C 85 E0  lea     ecx, [ebp+eax*4+var_20]
.text:0040933B 8B 11      mov     edx, [ecx]
.text:0040933D 8D 72 01    lea     esi, [edx+1]
.text:00409340 33 FF      xor    edi, edi
.text:00409342 3B F2      cmp    esi, edx
.text:00409344 72 05      jb    short loc_40934B
.text:00409346 83 FE 01    cmp    esi, 1
.text:00409349 73 03      jnb    short loc_40934E

.text:0040934B  loc_40934B:
.text:0040934B 33 FF      xor    edi, edi
.text:0040934D 47          inc    edi

.text:0040934E  loc_40934E:
.text:0040934E 89 31      mov    [ecx], esi
.text:00409350 8B CF      mov    ecx, edi
.text:00409352

.text:00409352  loc_409352:
.text:00409352 48          dec    eax
.text:00409353 79 DE      ins    short loc_409333

```

Figure 4: Subroutine reordering technique example.

4.4. Code Reordering through Jumps

Code reordering through jumps is based on the insertion of conditional or unconditional jumps to split a subroutine into two blocks of instructions. Afterwards, these blocks generated by the branching instruction are permuted to change the control flow. Cf. Figure 5. This technique was first employed by Win95/ZPerm family to generate new variants jointly with the insertion of garbage or dead code instructions.

5. Static Machine Learning Anti-Malware Tools

There are multiple ways in which malware can be represented from a static analysis point of view. Typically, features are

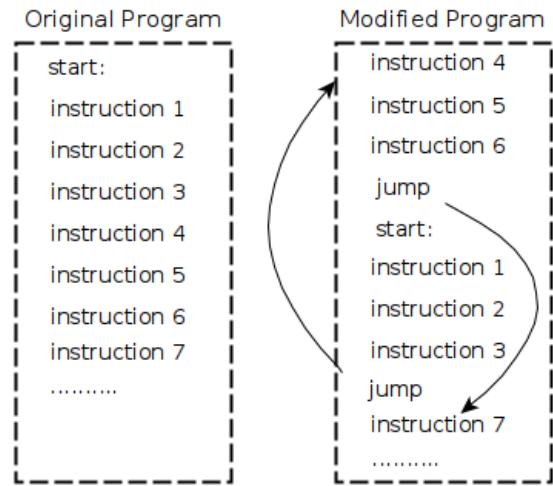


Figure 5: Code reordering through jumps example.

manually-engineered to capture some specific characteristics of the executable that can help distinguishing malware families or malware from benign software, e.g. API function calls, byte and opcode n-grams, etc (Souri and Hosseini, 2018; Ucci et al., 2019; Gibert et al., 2020b). In the present study, the machine learning approaches are limited to those whose performance would be affected by the functionality-preserving changes performed by the metamorphic attacks described in Section 4. Accordingly, the approaches assessed in this study can be divided in two groups: (1) n-gram based approaches or (2) deep learning approaches, depending on whether they take as input a feature vector containing an abstract representation of the executable or directly raw data.

5.1. N-gram based Approaches

An n-gram is a contiguous sequence of n items from a given sequence of text. N-grams can be extracted from the hexadecimal representation of malware’s binary content and from the assembly language source code. On the one hand, the hexadecimal representation represents the binary content of an executable as a sequence of bytes (base-16 number representation with digits [0-9] and [A-F]). Cf. Figure 6a. Alternatively, the assembly language source code contains the symbolic machine code of an executable with metadata information as function calls, memory allocation and variable information. Cf. Figure 6b. In consequence, byte n-grams (Zhang and Zhao, 2017; Raff and Nicholas, 2018) and opcode n-grams (Santos et al., 2013; Hu et al., 2013) refer to the unique combination of every n consecutive bytes and opcodes as individual features, respectively. An opcode refers to the name of a specific instruction, i.e. *ADD*, *MUL*, *PUSH*, etc, without its arguments.

N-gram based approaches construct a feature vector containing an abstract representation of malware, where each element in the vector indicates the number of appearances of a particular n-gram in the sequence of text. In consequence, the length of the feature vector depends on the number of unique n-grams, which increases with n . For instance, if we want to extract byte n-grams with $n = 3$, the number of possible n-grams is $256^3 = 16.777.216$. This leads to two main problems. First, the resulting feature vector is too large to keep in memory, even if malware n-grams do not increase exponentially with n but follow a Zipfian distribution (Raff et al., 2018c). Second, the machine learning model will be affected by the curse of dimensionality (Bellman, 2015; Chen, 2009) which means that the number

of samples in the dataset that need to be accessed to estimate a function with a given level of accuracy grows exponentially with the underlying dimensionality. As a result, methods in the literature reduced the high dimensional input space using feature selection techniques (Santos et al., 2013; Zhang and Zhao, 2017) or the hashing trick (Raff and Nicholas, 2018; Hu et al., 2013).

1. Feature selection is the process of selecting a subset of relevant features from the initial input space for use in model construction. A common approach is to rank the features based on the mutual information index in decreasing order (Santos et al., 2013; Zhang and Zhao, 2017). Mutual information, is an index of statistical dependence between two variables (Vergara and Estévez, 2014). In the case of a classification task, it measures the dependence between a feature X and the target variable Y . This is done by measuring how much knowing one of these variables reduces uncertainty about the other. The mutual information between two variables is a non-negative value, which measures the dependency between the variables. For two independent variables, their mutual information will be 0. Otherwise, for dependent variables, higher mutual information mean higher dependency.
2. Feature hashing, also known as the hashing trick, is a method for handling sparse, high-dimensional feature vectors by using a hash function to determine the feature’s location in a lower-dimensional vector. It can be seen as a random projection of the input space $A \in \mathbb{R}^n$ to a low dimensional space $B \in \mathbb{R}^m$, where $m \ll n$. More specif-

00401080	56	8D	44	24	08	50	88	F1	E8	1C	18	08	08	C7	06	88
00401080	BB	42	00	8B	C6	5E	C2	04	00	CC						
00401202	C7	01	08	BB	42	00	E9	26	1C	00	00	CC	CC	CC	CC	CC
00401306	56	88	F1	C7	06	BB	42	00	E8	13	1C	00	08	F6	44	
00401404	24	08	01	74	09	56	E8	6C	1E	00	08	C3	04	B8		
00401505	SE	C4	04	00	CC											
00401606	BB	44	24	08	8A	08	BB	54	24	04	88	A0	CC	CC	CC	CC
00401707	BB	44	24	04	8D	50	01	8A	08	49	84	C9	75	F9	2B	C2
00401808	C3	CC														
00401909	BB	44	24	10	8B	4C	24	08	BB	54	24	08	56	BB	74	24
00401A06	08	50	51	52	56	E8	18	1E	00	00	83	C4	10	BB	6C	5E
00401B08	C3	CC														
00401C00	BB	44	24	10	8C	4C	24	08	BB	54	24	08	56	BB	74	24
00401D00	08	50	51	52	56	E8	65	1E	00	00	83	C4	10	BB	6E	5E
00401E05	C3	CC														
00401F00	33	CC	C2	10	00	CC										
00401100	B8	88	00	00	00	C2	64	00	CC							
00401110	BB	03	00	00	00	C3	CC									
00401126	BB	08	00	00	00	C3	CC									
00401130	BB	44	24	04	A3	AC	49	52	00	BB	FE	FF	FF	FF	C2	04
00401140	00	CC														
00401150	AA	1C	AC	49	52	00	BB	C0	74	16	BB	48	24	08	BB	54
00401160	04	51	52	FF	D0	C7	05	AC	49	52	00	00	00	00	00	BB
00401170	FB	FF	FF	F2	C8	08	00	CC								
00401180	6A	04	68	00	10	00	00	68	68	BE	1C	00	6A	00	F5	15
00401190	9C	63	52	00	50	FF	15	C8	63	52	00	8B	4C	24	04	6A

(a) Hexadecimal view of a PE file. Each line is composed of the starting address of the machine codes in the memory and an accumulation of consecutive 16 byte values.

(b) Assembly view of the grayed part in Figure 6a. The first column represents the address, the second column the byte sequence and the third column the mnemonics sequence.

Figure 6: Hexadecimal and assembly view of a Portable Executable file.

ically, given an array of size N that counts the number of times each n-gram occurred, and a hash function, the hashing trick maps each n-gram to a location in the lower dimensional array.

Afterwards, the resulting low-dimensional feature vector is used for training a classification algorithm. In our experiments, we extracted 3-gram features from both the hexadecimal view and the assembly view. The high-dimensional feature vector was reduced using feature selection or the hashing trick. The size of the resulting low-dimensional vector is set to $K = 5000$ (different sizes for $K \in [500, 2000, 5000]$ were tried but $K = 5000$ provides the best performance). Afterwards, we trained various classifiers, including a logistic regression classifier and feed-forward neural networks consisting of 1 to 3 hidden layers. The number of neurons in the hidden layers was set to

2048, 1024 and 512 for the first, second and third hidden layer, respectively. The results presented in Section 6 show the performance of the best classifiers, which are referred using the following notation for the rest of the paper:

- *NN opcodes (hashing trick)* refers to a neural network with two hidden layers trained with the opcode-based feature vector reduced with the hashing trick technique. Cf. Figure 7.

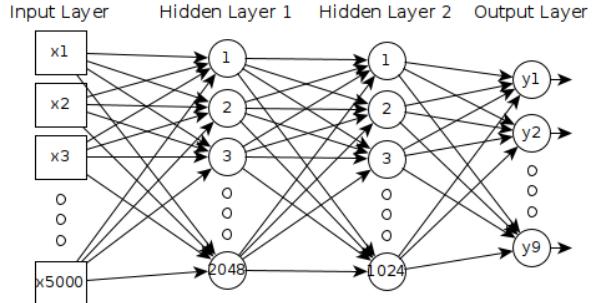


Figure 7: NN opcodes (hashing trick) architecture.

- *LR opcodes (mutual information)* refers to a logistic regression classifier trained with the top K opcode-based features selected using the mutual information index. Cf. Figure 8.

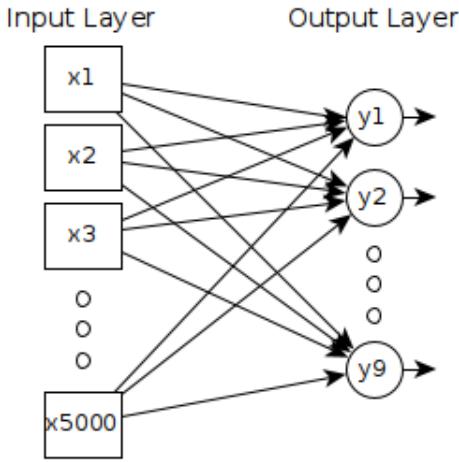


Figure 8: LR opcodes (mutual information) architecture.

- *NN bytes (hashing trick)* refers to a neural network with one hidden layer trained with the byte-based feature vector reduced with the hashing trick technique. Cf. Figure 9.

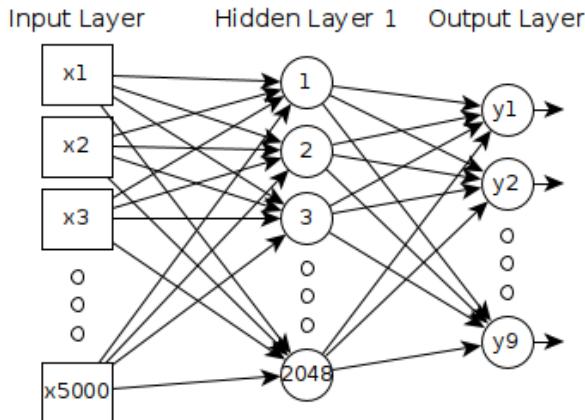


Figure 9: NN bytes (hashing trick) architecture.

Notice that feature selection using the mutual information metric has been only applied to the opcode-based 3-grams. This is because in the dataset used for training there are 55136 and 9514156 unique opcode and byte 3-grams, respectively, and in consequence, the memory requirements needed for selecting 5000 byte-based 3-gram features far exceeds the memory capacity of our system.

Instead of taking the number of appearances of n-grams as features, data normalization is applied to normalize the range of the features. The motivation behind data normalization is that the range of values of each feature varies widely. Machine learning models learn a mapping from input variables to an output variable. In consequence, the scale and distribution of the data drawn from the domain may be different for each variable or feature. For example, the number of times the 3-gram [mov, push, mov] or the 3-gram [pop, sub, and] may greatly differ. These differences in the scales across input variables may increase the difficulty of the problem being modeled. For instance, large input values can result in a model that learns large weight values, which are known to be often unstable, i.e. it may suffer from poor performance during learning and sensitivity to input values resulting in higher generalization error. Therefore, the features have been standarized so that they have zero-mean and unit-variance. The formula for standarization is as follows:

$$x' = \frac{x - \mu}{\delta}$$

where x is the original feature vector, μ is the mean of the feature values and δ is the standard deviation of the feature values.

5.2. Deep Learning Approaches

The need for manual feature engineering can be obviated by automated feature learning. Deep learning replaces the feature engineering process by an underlying system which typically consists of a neural network with multiple layers, that performs both feature learning and classification. With deep learning, one can start with raw data as features will be automatically created by the neural network when it learns. The main distinction between deep learning approaches for malware detection and classification lean on what they use as raw data.

5.2.1. Opcode-based Approaches

Opcode-based approaches (Gibert et al., 2017) take as input a sequence of assembly language instructions extracted from the assembly language source code of an executable. Gibert et al. (2017) proposed a shallow convolutional neural network to extract n-gram like features from malware’s instructions. This is achieved by a convolutional layer with filters of various sizes. In their work, an assembly program is represented as a concatenation of mnemonics

$$x_{1:n} = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

where n is the length of the program and $x_i \in \mathbb{R}^k$ corresponds to the i-th mnemonic in the program. Instead of representing the mnemonics as one-hot vectors, each mnemonic is represented as a word embedding. Following, a convolutional layer extracts n-gram like features. This is achieved by the convolution operator, which involves a filter $w \in \mathbb{R}^{hk}$ where h is the number of mnemonics to which is applied and k is the size of the word embedding. In particular, filters are applied to sequences containing from 3, 5 and 7 mnemonics. Cf. Figure A.18

5.2.2. Byte-based Approaches

Byte-based approaches (Raff et al., 2018a; Krčál et al., 2018; Gibert et al., 2018) are those that take as input a sequence of bytes extracted from the hexadecimal representation of the malware’s binary content. Cf. Figure 6a. These approaches face the following challenges:

- By treating an executable as a sequence of bytes, we are dealing with sequences of millions of time steps, which turns the task of malware detection and classification as one of the most challenging sequence classification problems with regard to the size of the time series (sequence of bytes).
- The meaning of any byte is dependent on its context and could encode any type of information, from binary code to human-readable text, images, etc.
- The same instruction could be encoded using different bytes depending on its arguments. For instance, the bytes sequence corresponding to the cmp instruction can begin with 0x3C, 0x3D, 0x3A, 0x3B, 0x80, 0x81, 0x38 or 0x39 depending on the arguments given.
- The content of a Portable Executable (PE) file exhibits various levels of spatial correlation. Nearby instructions within the same function are spatially correlated, but function calls and jcc instructions produce discontinuities over the code instructions and functions. As a result, this discontinuities are maintained through the bytes sequences.

Following it is provided a brief description of the architectures evaluated in the present study:

Raff et al. (2018a) proposed an architecture that consists of an embedding layer, a gated convolutional layer, a global max-pooling layer to produce its activations regardless of the location of the detected features, followed by fully-connected layers. This architecture will be called *MalConv* from now on. Cf. Figure A.19.

Krčál et al. (2018) presented a deep convolutional neural network architecture that consists of an embedding layer, four convolutions with strides and max-pooling between the second and third convolutions. Afterwards, it follows a global average pooling layer and various fully-connected layers. This architecture will be called *DeepConv* from now on. Cf. Figure A.20.

Gibert et al. (2018) presented a convolutional neural network architecture to categorize malware based on their structural entropy. The structural entropy of an executable is the representation of a file as a stream of entropy values, where each value describes the amount of entropy over a small chunk of code in a specific location of the file. Additionally, they proposed a multiresolution CNN to classify malware based on the approximation and details coefficients generated by the Haar wavelet transform over the entropy time series. The architectures will be called *Structural entropy CNN* and *Multiresolution CNN* from now on. Cf. Figures A.21, A.22.

5.2.3. Byte-based Shallow Convolutional Neural Network

Byte-based approaches presented in the literature (Raff et al., 2018a; Krčál et al., 2018; Gibert et al., 2018) tend to underperform in comparison to the opcode-based approaches (Gibert et al., 2017) as it can be observed in Section 6.3. Our intuition is that the size of their filters and the com-

plexity of the network architectures played a deep role. To check this premise, a shallow convolutional neural network architecture similar to the one presented by Gibert et al. (2017) has been proposed based on the raw byte sequences.

The architecture differs in the input of the network and in the size of the convolutional filters. Instead of receiving the assembly language instructions, the network takes as input the hexadecimal representation of the malware’s binary content represented as a concatenation of bytes:

$$x_{1:n} = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

where n is the length of the program and $x_i \in \mathbb{R}^k$ corresponds to the i -th byte in the program. The overall architecture is presented in Figure 10.

The network comprises the following layers:

- **Input layer.** The network takes as input a sequence of bytes, of size N , representing malware’s binary content.
- **Embedding layer.** Rather than perform convolutions on the raw byte values, each byte is mapped to a fixed length feature vector (word embedding) that is learnt during training. Take into account that using raw byte values would imply that certain byte values are intrinsically closer to each other than other byte values, which is known a priori to be false, as the meaning of the byte value is dependent on the context.
- **Convolutional layer.** This layer convolves various filters over the byte sequences and extracts n-gram like features from it. A convolution operation

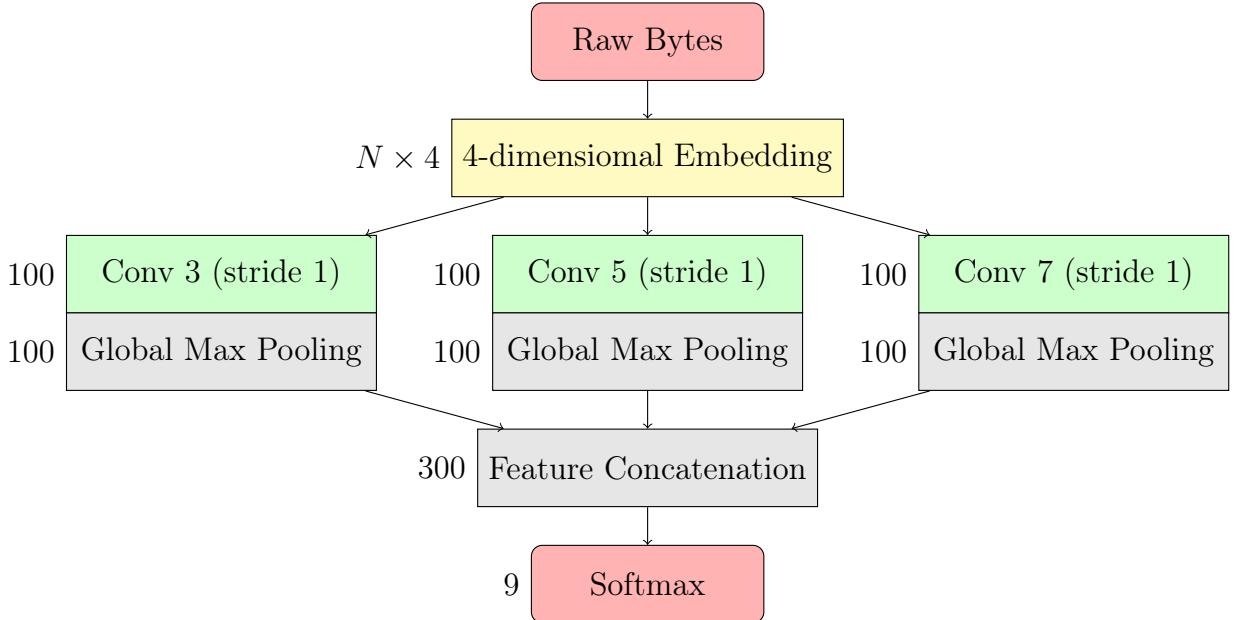


Figure 10: Convolutional neural network for malware classification from sequences of bytes.

involves a filter $w \in \mathbb{R}^{hk}$ where h is the number of bytes to which it is applied and k is the size of the word embedding. In particular, filters are applied to sequences containing 3,5 and 7 bytes.

A feature c_i is generated from a window of bytes $x_{i:i+h-1}$ (it comprises all bytes between position i and $i + h - 1$) and is defined as follows:

$$c_i = f(w \cdot x_{i:i+h-1} + b),$$

where f is a rectifier linear unit (ReLU) function and b the bias term.

- **Pooling layer.** Global max-pooling is applied to extract the maximum activation of each of the feature map activations generated by the convolutional layer.
- **Softmax layer.** It linearly combines the features learned by the previous

layers and applies the softmax function to generate a vector containing the normalized probability distribution over malware families.

Other variants of this architecture including dilated convolutions (Yu and Koltun, 2016) and gated linear units (Dauphin et al., 2017) have been evaluated but as it can be observed in Section 6.3, their performance is slightly worse than the standard convolution. These variants are named *Atrous CNN* and *CNN GLU*, respectively.

6. Evaluation

This section presents an extensive evaluation of the robustness of state-of-the-art detectors powered by ML against the metamorphic techniques presented in Section 4. The experiments are carried out using the data provided by Microsoft for the Big Data Innovators Gathering Challenge of 2015 (Ronen et al., 2018). Furthermore,

we investigate the utility of the aforementioned metamorphic techniques to augment the dataset and reduce class imbalance.

6.1. The Microsoft Malware Classification Challenge

Unlike other applications, the task of malware detection and classification has not received much attention in the research community and unfortunately, there are not available rich labeled datasets. Due to legal restrictions, benign binaries (e.g. executables of common Windows applications, utilities, etc) can not be shared, as they are often protected by copyright laws. Contrarily, there are websites such as VirusShare and VXHeaven that share malicious executables. However, unlike other domains where data may be labeled very rapidly by a non-expert, determining whether a file is malicious and its corresponding family or class is a very time-consuming process, even for security experts. In consequence, for reproducibility purposes the research conducted in this paper has been evaluated on the data provided by Microsoft for the Big Data Innovators Gathering Challenge of 2015 (Ronen et al., 2018), which over the years has become the de facto benchmark for evaluating approaches on the task of malware classification. Microsoft provided a high-quality public labeled benchmark of almost half a terabyte of malware. Nowadays, the dataset is hosted on Kaggle⁶ and is publicly accessible. The dataset contains samples of malware representing a mix of 9 malware families (See Table 1): (1) RAMNIT, (2) LOLLIPOP, (3) KELIHOS_VER3, (4) VUNDO, (5) SIMDA,

(6) TRACUR, (7) KELIHOS_VER1, (8) OB-FUSCATOR.ACY and (9) GATAK.

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

6.2. Experimental Setup

The experiments have been carried out on a machine with an Intel Core i7-7700K CPU, 4xGeforce GTX 1080Ti and 64Gb of RAM. All algorithms have been implemented using TensorFlow (Abadi et al., 2015).

The experimentation has been divided into two phases. The first phase analyses the individual impact of the metamorphic techniques on the performance of the machine learning models. Oppositely, the second phase analyzes the usage of the metamorphic techniques for augmenting the training set and boosting the performance of the machine learning classifiers.

6.3. Analysis of the Performance of ML Classifiers against Metamorphic Techniques

To analyze the performance of the ML classifiers, the dataset has been divided into three sets: (1) the training set, (2) the validation set and (3) the test set, containing 70%, 15% and 15% of the samples, respectively.

Instead of the accuracy, the macro F1-score has been used to evaluate the models. This is due to the fact that accuracy alone

⁶<https://www.kaggle.com/c/malware-classification/>

can be a misleading measure, specially in datasets with large class imbalance. For instance, a machine learning model might correctly predict the value of the majority class for all predictions and achieve a high classification accuracy while failing to correctly predict the class of samples belonging to the minority and critical classes. In this situation the macro F1-score metric is more adequate because it penalizes this kind of behavior by calculating the unweighted mean of the precision and recall for each label or class. The mathematical formulation of the macro f1-score is as follows:

$$\text{macro } f1 \text{ score} = \frac{1}{q} \sum_{i=1}^q F_1^i$$

where F_1^i is the weighted average of precision and recall in class i .

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R}$$

When evaluating the performance of the classifiers, the test set is used to generate three obfuscated test sets (A, B and C) and the average macro f1-score achieved on the three sets is provided as outcome.

6.3.1. Dead Code Insertion

To evaluate the resilience of ML models to the changes performed on the code by the dead code insertion technique, the test set has been obfuscated by inserting 10, 50, 100, 200 and 500 dead code instructions in random positions within the assembly language source code of executables (See Section 4.1). This includes all sections that contain assembly language instructions, and not only the .text section. Given $P = [0.5, 0.071, \dots, 0.071]$ and $X = [NOP, MOV Reg Reg, \dots]$, where $|P|$ and $|X|$ equals N , $P(n)$ is the probability to

select the $X(n)$ element. Accordingly, the probability to insert the *NOP* instruction is 0.5 while the probability to insert any other instruction is 0.071. The probability values have been set as described above to put more focus on the *NOP* instruction.

Table 2: Comparison of the macro f1-score achieved by the ML classifiers on the test set obfuscated by the dead code insertion technique.

	Method	Test Set	Dead code insertion					
			10	50	100	200	500	
Opcode-based	(Gibert et al., 2017)		0.9852	0.9806	0.9794	0.9711	0.9672	0.8754
	NN opcodes (hashing trick)		0.9765	0.9754	0.9738	0.9704	0.9458	0.7031
Byte-based	NN opcodes (mutual information)		0.9876	0.9873	0.9888	0.9887	0.9879	0.9728
	MalConv (Raff et al., 2018a)		0.8480	0.8543	0.8502	0.8497	0.8300	0.8328
DeepConv (Krcal et al., 2018)	Structural entropy CNN (Gibert et al., 2018)		0.8376	0.8030	0.7913	0.7567	0.6944	0.6765
	Multiresolution CNN (Gibert et al., 2018)		0.8892	0.8952	0.9015	0.8880	0.8348	0.7818
NN bytes (hashing trick)	NN bytes (hashing trick)		0.8972	0.9074	0.9008	0.8880	0.8297	0.7995
	NN bytes (mutual information)		0.8858	0.8863	0.8854	0.8864	0.8863	0.8861
Shallow CNN	Shallow CNN		0.9748	0.9733	0.9731	0.9694	0.9674	0.9577
	Dilated CNN		0.9622	0.9369	0.9362	0.9427	0.9317	0.9280
CNN GLU	CNN GLU		0.9627	0.9627	0.9627	0.9623	0.9606	0.9638

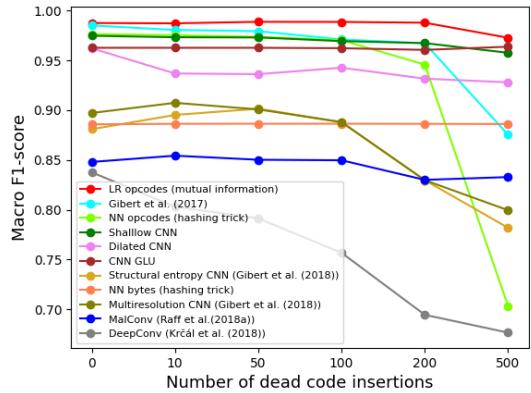


Figure 11: Macro f1-score of ML classifiers on the test set obfuscated by the dead code insertion technique.

Table 2 and Figure 11 present the classification performance of the ML classifiers against the aforementioned obfuscated test set. It can be observed that the performance of the opcode-based methods (Gibert et al. (2017), NN opcodes (hashing trick)) degrade considerably when more than 200 dead code instructions are inserted per sample. On the contrary, byte-based classifiers (Raff et al. (2018a); Krčál et al. (2018); Gibert et al. (2018), NN bytes (hashing

trick)) remain more stable to the changes in the executable. There are two reasons for this occurrence: (1) First, the size of the executables significantly differ between families (Gibert et al., 2020a). The less opcodes has the sample the easy is to obfuscate the patterns learned by the ML model; (2) Second, samples belonging to some families in the dataset do not contain any NOP instruction in their assembly language source code such as samples belonging to the Kelihos_ver1 family. This have caused the opcode-based models to learn that if there exist a n-gram containing the NOP instruction in the assembly language source code, the corresponding executable might belong to any other family but not to Kelihos_ver1. For instance, as it can be observed in Figure 12 the opcode-based CNN (Gibert et al., 2017) only classified correctly 27 out of 67 samples of the Kelihos_ver1 families. To check this premise, in Section 6.5 it is augmented the training data by using a combination of the metamorphic techniques presented in Section 4, including the dead code insertion technique. Results show that the degradation in the performance of opcode-based classifiers is due to data bias rather than to any weaknesses of the opcode-based classifiers with respect to the byte-based classifiers and could be resolved by augmenting the training set with some samples of those families containing some NOP instructions.

6.3.2. Register’s Reassignment

The second metamorphic technique evaluated is the register’s reassignment technique (See Section 4.2). To evaluate the resilience of ML models against the register’s reassignment technique, two or more data registers (i.e. *EAX*, *EBX*, *ECX*, *EDX*) of the samples in the test set have been

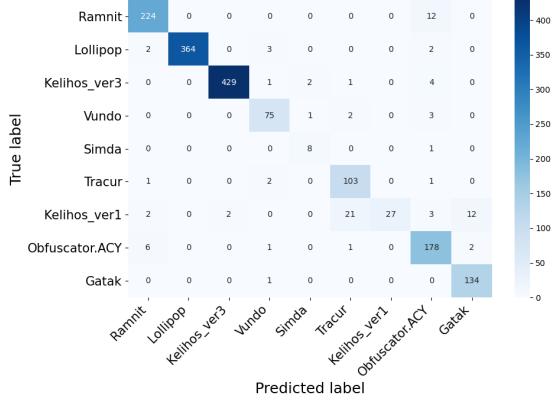


Figure 12: Opcode-based CNN confusion matrix (500 dead code insertions).

swapped. More specifically, two experiments have been performed:

- Experiment A: Swap two randomly selected data registers.
- Experiment B: Swap all data registers.

Table 3: Comparison of the macro f1-score achieved by the ML classifiers on the test set obfuscated by the register’s reassignment technique.

Method	Test Set	Register’s Reassignment	
		A	B
MalConv (Raff et al., 2018a)	0.8480	0.8465	0.8294
DeepConv (citepkrčál2018deep)	0.8376	0.8136	0.7768
Structural entropy CNN (Gibert et al., 2018)	0.8809	0.8850	0.8954
Multiresolution CNN (Gibert et al., 2018)	0.8972	0.8973	0.8996
NN bytes (hashing trick)	0.8858	0.8846	0.8764
Shallow CNN	0.9748	0.9708	0.9565
Dilated CNN	0.9622	0.9410	0.9308
CNN GLU	0.9627	0.9638	0.9570

Table 3 and Figure 13 present the performance of the ML classifiers against the samples of the test set obfuscated with the register’s reassignment technique. Notice that opcode-based approaches are not evaluated as they are not affected by this technique. On the contrary, Malconv’s (Raff et al., 2018a), DeepConv’s (Krčál et al., 2018), the shallow CNN and the n-gram based classifier’s performance degraded 2.19%, 7.26%, 1.88% and 1.06% respectively while the

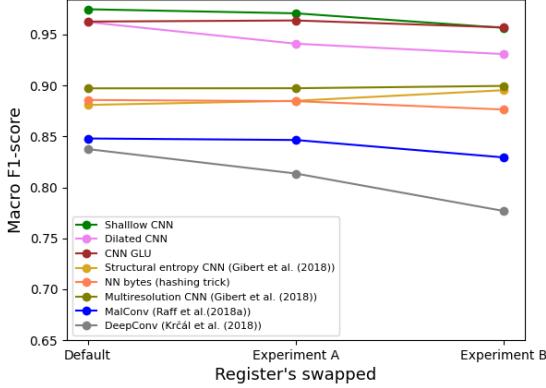


Figure 13: Macro f1-score of ML classifiers on the test set obfuscated by the register’s reassignment technique.

macro f1-score of the methods presented in Gibert et al. (2018) slightly increased. Our intuition is that both classifiers (Gibert et al., 2018) as they are based on the structural entropy of an executable, even that the register reassignment technique replaces some bytes in the executable, the entropy time series remain mostly unaltered. In addition, it can be observed that the performance of the classifiers decreases as the number of registers swapped augments.

6.3.3. Subroutine Reordering

To evaluate the resilience of ML models to the modifications performed on the code by the subroutine reordering technique, the samples on the test set have been obfuscated by performing 5, 10, 20 and 50 random subroutine permutations (See Section 4.3).

Table 4 and Figure 14 display the performance of the ML models over the obfuscated test set. It can be observed that the classification performance of all models remain mostly constant. This is because most neural network architectures evaluated contain a global max-pooling (Raff et al., 2018a; Gibert et al., 2017) or global

Table 4: Comparison of the macro f1-score achieved by the ML classifiers on the test set obfuscated by the subroutine reordering technique.

	Method	Test Set	Subroutine reorderings			
			5	10	20	50
Opcode-based	Gibert et al., 2017)	0.9852	0.9807	0.9801	0.9798	0.9784
	NN opcodes (hashing trick)	0.9765	0.9784	0.9781	0.9785	0.9783
	NN opcodes (mutual information)	0.9876	0.9876	0.9876	0.9874	0.9874
Byte-based	MalConv (Raff et al., 2018a)	0.8480	0.8558	0.8391	0.8645	0.8552
	DeepConv (Krčál et al., 2018)	0.8376	0.8025	0.8150	0.7955	0.7903
	Structural entropy CNN (Gibert et al., 2018)	0.8809	0.8976	0.8929	0.9026	0.8990
	Multiresolution CNN (Gibert et al., 2018)	0.8972	0.9044	0.8998	0.9013	0.9008
	NN bytes (hashing trick)	0.8858	0.8859	0.8860	0.8852	0.8851
	Shallow CNN	0.9748	0.9732	0.9733	0.9731	0.9734
	Dilated CNN	0.9622	0.9465	0.9418	0.9474	0.9560
	CNN GLU	0.9627	0.9623	0.9627	0.9626	0.9623

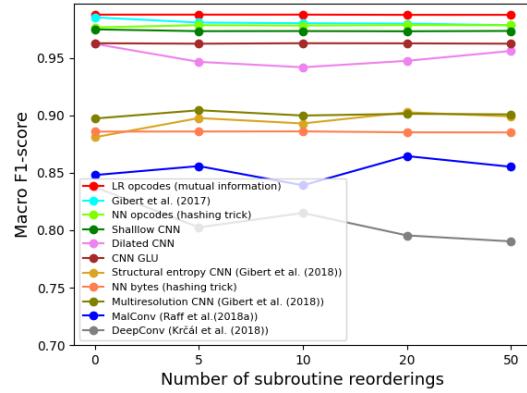


Figure 14: Macro f1-score of ML classifiers on the test set obfuscated by the subroutine reordering technique.

avg-pooling layer (Krčál et al., 2018) at the end of the convolutional layers which allowed the detection of patterns independently of their position in the raw input sequences.

6.3.4. Code Reordering through Jumps

The resilience of the ML models against the code reordering through jumps technique (See Section 4.4) is assessed by reordering the code with the insertion of 5, 10, 20 and 50 jumps randomly within the assembly language source code of the executables.

Table 5 and Figure 15 present the performance of the ML classifiers against the samples of the test set obfuscated with the code reordering through jumps technique.

Table 5: Comparison of the macro f1-score achieved by the ML classifiers on the test set obfuscated by the code reordering through jumps technique.

	Method	Test Set	Code reordering through jumps			
			5	10	20	50
Opcode-based	(Gibert et al., 2017)		0.9852	0.9789	0.9776	0.9768
	NN opcodes (hashing trick)		0.9765	0.9784	0.9777	0.9765
	NN opcodes (mutual information)		0.9876	0.9874	0.9876	0.9876
Byte-based	MalConv (Raff et al., 2018a)		0.8480	0.8482	0.8573	0.8374
	DeepConv (Krčál et al., 2018)		0.8376	0.7547	0.7243	0.7136
	Structural entropy CNN (Gibert et al., 2018)		0.8809	0.8416	0.8162	0.7671
	Multiresolution CNN (Gibert et al., 2018)		0.8972	0.8453	0.8195	0.7798
	NN bytes (hashing trick)		0.8858	0.8841	0.8954	0.8907
	Shallow CNN		0.9748	0.9732	0.9739	0.9733
	Dilated CNN		0.9622	0.9503	0.9535	0.9605
	CNN GLU		0.9627	0.9625	0.9628	0.9627

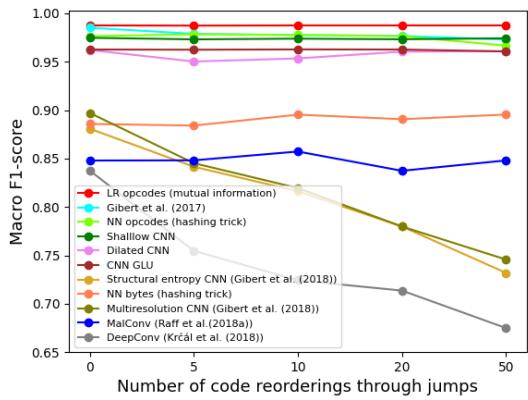


Figure 15: Macro f1-score of ML classifiers on the test set obfuscated by the code reordering through jumps technique.

Results are similar to Section 6.3.3 with the byte-based approaches performing poorly in comparison to opcode-based approaches, and additionally, the performance of DeepConv (Krčál et al., 2018), structural entropy and haar approximation & coefficients (Gibert et al., 2018) degraded considerably from 0.8376, 0.8809, 0.8972 to 0.6749, 0.7319, 0.7457, respectively. At the present moment we don't have an explanation for the behavior of the aforementioned models but it might be the case that the code reordering through jumps technique modifies the initial byte sequences in such a way that the resulting time series look completely different from the non-obfuscated versions. For instance, in Figure 16 it can be ob-

served that the structural entropy visualization of the malicious sample with ID *bxEED6RSpmnWV03kyMLoK* diverges from the entropy representation of the version obfuscated by reordering the source code with the random insertion of 50 jumps. Thus, if the modifications alter the byte sequence in a way that the patterns learned by the ML classifiers do not occur, then the sample will be misclassified. In addition, the higher complexity of the aforementioned architectures have negatively affected its performance. On the contrary, it can be observed in Figure 15 that the shallow-based CNN presented in Section 5.2.3 is resilient to the changes performed by the code reordering through jumps technique.

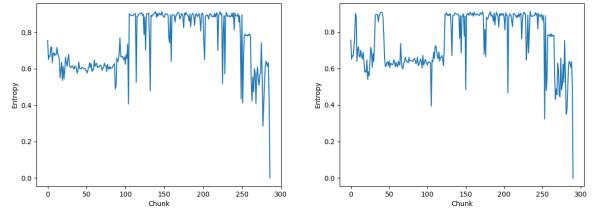


Figure 16: Structural entropy comparison between the non-obfuscated and obfuscated versions of the malicious sample with ID *bxEED6RSpmnWV03kyMLoK*.

6.3.5. Mixed Obfuscation

Finally, the samples in the test set have been altered by various combinations of the four metamorphic techniques to mimic the changes performed by a metamorphic engine. To this end, four experiments have been performed:

- Experiment A: Each sample in the test set has been modified by inserting 10 dead code insertions, performing 10 subroutine reorderings, 10 code reorderings through jumps, and by swapping all four data registers.

- Experiment B: Each sample in the test set has been modified by inserting 50 dead code insertions, performing 20 subroutine reorderings, 20 code reorderings through jumps, and by swapping all four data registers.
- Experiment C: Each sample in the test set has been modified by inserting 100 dead code insertions, performing 30 subroutine reorderings, 30 code reorderings through jumps, and by swapping all four data registers.
- Experiment D: Each sample in the test set has been modified by inserting 200 dead code insertions, performing 40 subroutine reorderings, 40 code reorderings through jumps, and by swapping all four data registers.

Table 6: Comparison of the macro f1-score achieved by the ML classifiers on the test set obfuscated with various metamorphic techniques.

	Method	Test Set	Experiment		
			A	B	C
Opcode-based	(Gibert et al., 2017)	0.9852	0.9754	0.9718	0.9632
	NN opcodes (hashing trick)	0.9765	0.9776	0.9756	0.9618
	NN opcodes (mutual information)	0.9876	0.9880	0.9889	0.9878
Byte-based	MalConv (Raff et al., 2018a)	0.8480	0.8496	0.8340	0.8353
	DeepConv (Krčál et al., 2018)	0.8376	0.6981	0.6746	0.6738
	Structural entropy CNN (Gibert et al., 2018)	0.8809	0.7988	0.7674	0.7360
	Multiresolution CNN (Gibert et al., 2018)	0.8972	0.8215	0.7872	0.7533
	NN bytes (hashing trick)	0.8858	0.8715	0.8744	0.8842
	Shallow CNN	0.9748	0.9576	0.9499	0.9652
	Dilated CNN	0.9622	0.9291	0.9376	0.9331
	CNN GLU	0.9627	0.9578	0.9569	0.9567

Table 6 and Figure 17 show the performance of the ML classifiers on the obfuscated test sets. Similarly to the previous experiments, opcode-based approaches achieve better results than byte-based approaches. In addition, it can be observed that approaches based on the manual extraction of n-gram features are more resilient to metamorphic techniques as they are mostly unaffected by them. On the other hand, from those approaches based on deep learning, the opcode-based CNN (Gibert et al., 2017) is the one that achieved

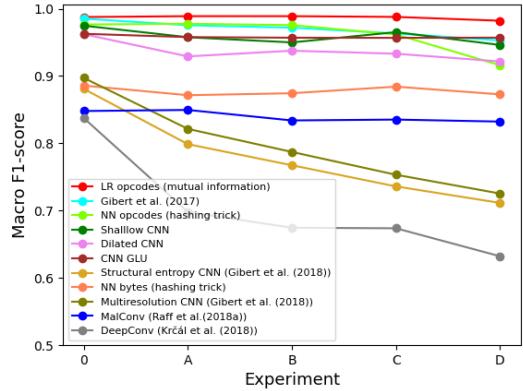


Figure 17: Macro f1-score of ML classifiers on the test set obfuscated with various metamorphic techniques.

the highest macro f1-score while DeepConv (Krčál et al., 2018) is the ML model that was most negatively affected by the modifications in the malware’s code. Our intuition is that the size of its filters and the complexity of the network, jointly with the imbalanced data played a deep role.

6.4. Summary

Table 7 compares the performance of the ML classifiers on the obfuscated test sets. To sum up, opcode-based approaches (Gibert et al. (2017), NN opcodes (hashing trick), LR opcodes (mutual information)) perform considerably better than byte-based approaches (Raff et al. (2018a); Krčál et al. (2018); Gibert et al. (2018), NN bytes (hashing trick)), with the logistic regression classifier achieving the highest macro f1-score in all the obfuscated sets. The strong performance of n-gram features have already been investigated in the literature, e.g. Zhang et al. (2016) and the Winner’s solution of the Microsoft Malware Classification Challenge ⁷, in which proved

⁷https://github.com/xiaozhouwang/kaggle_Microsoft_Malware/tree/master/

Table 7: Macro F1-score achieved by ML classifiers on the obfuscated test set.

Method	Test Set	Dead code insertion					Register's reassignment		Subroutine reorderings					Code reordering through jumps					Mixed			
		10	50	100	200	500	A	B	5	10	20	50	A	B	C	D	A	B	C	D		
Opcodes-based	(Gibert et al., 2017)	0.9852	0.9806	0.9794	0.9711	0.9672	0.8754	0.9852	0.9852	0.9807	0.9801	0.9798	0.9784	0.9789	0.9776	0.9768	0.9729	0.9754	0.9718	0.9632	0.9529	
	NN opcodes (hashing trick)	0.9765	0.9754	0.9738	0.9704	0.9458	0.7031	0.9765	0.9765	0.9784	0.9781	0.9785	0.9783	0.9784	0.9777	0.9765	0.9667	0.9776	0.9756	0.9618	0.9159	
	NN opcodes (mutual information)	0.9876	0.9873	0.9888	0.9887	0.9879	0.9728	0.9876	0.9876	0.9876	0.9876	0.9874	0.9874	0.9874	0.9876	0.9876	0.9876	0.9889	0.9889	0.9878	0.9821	
Byte-based	MalConv (Raff et al., 2018a)	0.8480	0.8543	0.8502	0.8497	0.8300	0.8328	0.8465	0.8294	0.8558	0.8391	0.8645	0.8552	0.8482	0.8573	0.8374	0.8481	0.8496	0.8340	0.8353	0.8323	
	DeepConv (Krčál et al., 2018)	0.8376	0.8030	0.7913	0.7567	0.6944	0.6765	0.8136	0.7768	0.8025	0.8150	0.7955	0.7903	0.7547	0.7243	0.7136	0.6749	0.6981	0.6746	0.6738	0.6322	
	Structural entropy CNN (Gibert et al., 2018)	0.8972	0.9074	0.9008	0.8880	0.8297	0.7995	0.8973	0.8996	0.9044	0.8998	0.9013	0.9008	0.8453	0.8195	0.7798	0.7457	0.8215	0.7872	0.7533	0.7254	
	Multiresolution CNN (Gibert et al., 2018)	0.8858	0.8863	0.8854	0.8864	0.8863	0.8861	0.8846	0.8764	0.8859	0.8860	0.8852	0.8851	0.8841	0.8954	0.8907	0.8955	0.8715	0.8744	0.8842	0.8727	
	NN bytes (hashing trick)	0.9748	0.9733	0.9731	0.9694	0.9674	0.9577	0.9708	0.9565	0.9732	0.9733	0.9731	0.9734	0.9732	0.9739	0.9733	0.9743	0.9576	0.9499	0.9652	0.9461	
	Shallow CNN	0.9622	0.9369	0.9362	0.9427	0.9317	0.9280	0.9410	0.9308	0.9465	0.9418	0.9474	0.9560	0.9563	0.9535	0.9605	0.9609	0.9291	0.9376	0.9331	0.9216	
	Dilated CNN	0.9627	0.9627	0.9627	0.9623	0.9606	0.9638	0.9638	0.9570	0.9623	0.9627	0.9626	0.9623	0.9625	0.9628	0.9627	0.9607	0.9579	0.9569	0.9567	0.9570	

to be decisive features for the construction of their classifier.

On the other hand, although deep learning approaches have shown great adoption in recent years by the cybersecurity industry, they are still in an early stage and there still exist margin for improvement. As observed in Table 7, the performance of the deep learning approaches varies greatly depending on the input of the network. For instance, the opcode-based shallow model (Gibert et al., 2017) performance degraded considerably when 500 random dead code instructions were inserted within the source code of the samples in the test set mainly because some bias in the dataset. Regarding byte-based approaches, those that take as input the raw byte sequences (Raff et al., 2018a; Krčál et al., 2018) are mostly affected by the register reassignment technique while entropy-based approaches (Gibert et al., 2018) demonstrated robustness against it. Furthermore, the performance of all deep learning approaches, indistinctly of their input, show some degradation with respect to the changes performed by the code reordering through jumps technique. In addition, it can be observed that there exist a huge gap in the performance of opcode-based (Gibert et al., 2017) and byte-based approaches (Raff et al., 2018a; Krčál et al., 2018; Gibert et al., 2018) deep learning

methods in the literature, achieving a macro f1-score on the test set equals to 0.9852, 0.8480, 0.8376, 0.8972, respectively. This is because byte-based approaches failed to correctly classify samples belonging to the minority classes, e.g. Simda, Obfuscator.ACY, etc. This is attributable to several factors: (1) the complexity and depth of the network architectures; (2) the size of the filters and (3) class imbalance. As it is shown in Table 7, the proposed shallow architecture trained on the raw bytes sequences with filters of various sizes ranging from $k \in \{3, 5, 7\}$ achieves a macro f1-score comparable to the opcode-based approaches and 14.95% and 16.38% higher than MalConv’s (Raff et al., 2018a) and DeepConv’s (Krčál et al., 2018) architectures, respectively. Thus, it demonstrates that the complexity of the network architectures played an important role in the low output achieved by the byte-based models. In addition, the byte-based shallow CNN architecture has shown greater robustness against the dead code insertion technique than their opcode-based counterpart although it is still affected by the register’s reassignment technique as all byte-based approaches.

6.5. Data Augmentation with Adversarial Examples

Recent advances in deep learning have been largely attributed to the quantity and diversity of data. The more data and the

more variation possible in the data the better the generalization of the model will be. However, in some cases it is not possible to collect thousands or millions of samples. In such cases, more data can be generated from a given dataset. This process is known as data augmentation. As far as we know, no data augmentation scheme has been proposed in the literature for the malware domain. Following, the use of metamorphic techniques for augmenting the dataset is investigated. The main idea behind using the aforementioned metamorphic techniques to augment the dataset is that the modifications preserve the functionality of the executables and are commonly used by malware authors and thus, it may help build robust ML models. As observed in Table 1 the training set is very imbalanced, with the majority class (Kelihos_ver3) containing 70 times more samples than the minority class (Simda). Subsequently, the number of samples generated for each family varied considerably, with the samples of the minority families reused more than the samples in the majority families to expand the training set. The total number of samples in the training set is shown in Table 8. The augmented training set contains the original sample and one or more obfuscated versions of it. This obfuscated versions have been generated using the following parameters:

- A total of 10 dead code instructions have been inserted.
- 5 subroutines have been randomly permuted.
- The source code has been reordered by inserting 5 jumps to split the subroutines.
- The registers of a given sample have

been randomly swapped with probability $p = 0.2$.

Table 8: Class distribution in augmented training set.

Family Name	Training set	Augmented training set
Ramnit	1084	3249
Lollipop	1736	3472
Kelihos_ver3	2057	4114
Vundo	327	2289
Simda	26	546
Tracur	532	2660
Kelihos_ver1	279	2511
Obfuscator.ACY	845	3380
Gatak	721	2884

Table 9 presents the performance of the models trained using the augmented training set on the obfuscated test set. In general, all deep learning approaches gained some robustness against the changes performed by the metamorphic techniques. It can be observed that byte-based approaches improved considerably thanks to the augmented training set. For instance, MalConv’s (Raff et al., 2018a) and DeepConv’s (Krčál et al., 2018) performance improved 9.98% and 5.03%, respectively. However, their models continue to perform poorly in comparison to the byte-based shallow CNN (0.9748 macro f1-score on the test set), the dominant byte-based classifier. On the other hand, the robustness of the opcode-based CNN (Gibert et al., 2017) improved considerably with respect to the modifications performed by the dead code insertion technique. The macro f1-score achieved by the opcode-based CNN on the test set obfuscated by inserting 500 dead code instructions improved from 0.8754 to 0.9796, an 11.9% increase.

Notice that the macro f1-score achieved by the shallow CNN approaches, indistinctly of whether they take as input the bytes or opcode sequences, is close to those

Table 9: Macro F1-score achieved by ML classifiers on the obfuscated test set trained on the augmented dataset (AD).

	Method	Test Set	Dead code insertion					Register’s Reassignment			Subroutine reorderings					Code reordering through jumps					Mixed			
			10	50	100	200	500	A	B		5	10	20	50		5	10	20	50	A	B	C	D	
Opcode-based	(Gibert et al., 2017)		0.9852	0.9806	0.9794	0.9711	0.9672	0.8754	0.9852	0.9852	0.9807	0.9801	0.9798	0.9784	0.9789	0.9776	0.9768	0.9729	0.9754	0.9718	0.9632	0.9529		
	(Gibert et al., 2017), AD		0.9850	0.9854	0.9835	0.9855	0.9836	0.9796	0.9850	0.9850	0.9849	0.9848	0.9845	0.9850	0.9835	0.9821	0.9811	0.9781	0.9821	0.9791	0.9787	0.9767		
	NN opcodes (hashing trick)		0.9765	0.9754	0.9738	0.9704	0.9458	0.7031	0.9765	0.9765	0.9784	0.9781	0.9785	0.9783	0.9784	0.9777	0.9765	0.9667	0.9776	0.9756	0.9618	0.9159		
	NN opcodes (hashing trick, AD)		0.9862	0.9827	0.9833	0.9734	0.9298	0.7659	0.9862	0.9862	0.9862	0.9862	0.9862	0.9862	0.9859	0.9858	0.9859	0.9820	0.9857	0.9820	0.9684	0.9103		
	LR opcodes (mutual information)		0.9876	0.9873	0.9888	0.9887	0.9879	0.9728	0.9876	0.9876	0.9876	0.9874	0.9874	0.9874	0.9876	0.9876	0.9876	0.9876	0.9889	0.9889	0.9878	0.9821		
	LR opcodes (mutual information, AD)		0.9873	0.9875	0.9876	0.9852	0.9858	0.9836	0.9873	0.9873	0.9878	0.9875	0.9875	0.9880	0.9873	0.9875	0.9875	0.9875	0.9875	0.9875	0.9865	0.9865		
Byte-based	MalConv (Raff et al., 2018a)		0.8480	0.8543	0.8502	0.8497	0.8300	0.8328	0.8465	0.8294	0.8558	0.8391	0.8645	0.8552	0.8482	0.8573	0.8374	0.8481	0.8496	0.8340	0.8353	0.8323		
	MalConv (Raff et al., 2018a), AD		0.9326	0.9483	0.9399	0.9434	0.9398	0.9422	0.9339	0.9388	0.9427	0.9415	0.9404	0.9393	0.9341	0.9387	0.9404	0.9453	0.9367	0.9346	0.9335	0.9280		
	DeepConv (Krčál et al., 2018)		0.8376	0.8030	0.7913	0.7567	0.6944	0.6765	0.8136	0.7768	0.8025	0.8150	0.7955	0.7903	0.7547	0.7243	0.7136	0.6749	0.6981	0.6746	0.6738	0.6322		
	DeepConv (Krčál et al., 2018), AD		0.8707	0.8812	0.8809	0.8680	0.8585	0.8252	0.8762	0.8712	0.8801	0.8802	0.8760	0.8741	0.8653	0.8653	0.8513	0.8523	0.8541	0.8474	0.8409	0.8388		
	Structural entropy CNN (Gibert et al., 2018)		0.8809	0.8952	0.9015	0.8888	0.8348	0.7518	0.8850	0.8954	0.8976	0.8929	0.9026	0.8990	0.8416	0.8162	0.7671	0.7319	0.7984	0.7674	0.7360	0.7115		
	Structural entropy CNN (Gibert et al., 2018), AD		0.8904	0.8906	0.9022	0.8941	0.9223	0.8904	0.8988	0.8941	0.8987	0.8970	0.9012	0.9019	0.9023	0.8911	0.8796	0.8759	0.8806	0.8887	0.8744	0.8678		
Byte-based	Multiresolution CNN (Gibert et al., 2018)		0.8972	0.9074	0.9008	0.8880	0.8297	0.7995	0.8973	0.8996	0.9044	0.8996	0.9013	0.9008	0.8453	0.8105	0.7798	0.7457	0.8215	0.7872	0.7533	0.7254		
	Multiresolution CNN (Gibert et al., 2018), AD		0.9261	0.9244	0.9256	0.9305	0.9286	0.8335	0.9264	0.9319	0.9241	0.9233	0.9282	0.9244	0.9182	0.9215	0.897	0.8954	0.9179	0.9052	0.8945	0.8784		
	NN bytes (hashing trick)		0.8858	0.8863	0.8854	0.8864	0.8863	0.8861	0.8846	0.8764	0.8859	0.8860	0.8852	0.8851	0.8841	0.8954	0.8907	0.8955	0.8715	0.8744	0.8842	0.8727		
	NN bytes (hashing trick, AD)		0.9539	0.9498	0.9474	0.9492	0.9492	0.9465	0.9410	0.9363	0.9491	0.9522	0.9494	0.9489	0.9491	0.9437	0.9464	0.9403	0.9439	0.9349	0.9448	0.9360		
	Shallow CNN		0.9748	0.9733	0.9731	0.9694	0.9674	0.9577	0.9708	0.9565	0.9732	0.9733	0.9731	0.9734	0.9732	0.9739	0.9733	0.9743	0.9576	0.9499	0.9652	0.9461		
	Shallow CNN, AD		0.9765	0.9762	0.9740	0.9744	0.9745	0.9609	0.9723	0.9748	0.9763	0.9762	0.9760	0.9712	0.9765	0.9769	0.976	0.9753	0.9690	0.9734	0.9718	0.9618		

obtained by the n-gram based approaches. In the case of the byte-based CNN it achieves higher macro f1-score than the n-gram based approach while the opcode-based CNN macro f1-score is marginally lower than the n-gram based counterpart. Thus, demonstrating that deep learning approaches are a good alternative to n-gram based approaches without the computational and memory costs of having to exhaustively enumerate millions of features and manually perform feature extraction and selection during training.

7. Conclusions & Future Work

This paper provides an exhaustive evaluation of the vulnerability of state-of-the-art anti-malware engines to the changes in the source code generated by the following metamorphic techniques: (1) the dead code insertion technique, (2) the register’s reassignment technique, (3) the subroutine reordering technique and (4) the code reordering through jumps technique. Results show that byte-based approaches perform poorly in comparison with opcode-based approaches and are not robust to the changes caused by the register’s reassignment technique. Their lower yield is attributable to several factors: (1) the size of the filters, (2) the complexity of the net-

work and (3) the data imbalance. On the other hand, the shallow architecture presented in this paper has achieved an improvement of 14.95% and 16.38% with respect to MalConv’s (Raff et al., 2018a) and DeepConv’s (Krčál et al., 2018) architectures, respectively, and attains similar classification performance in comparison to opcode-based approaches (Gibert et al., 2017). Furthermore, the usage of metamorphic techniques to augment the training set has been investigated. Results show that the classification performance of deep learning approaches improves considerably and gain robustness against metamorphism independently of their input data. Thus, we demonstrate the feasibility of augmenting the training data, and in particular the number of samples belonging to the minority classes, by employing metamorphic techniques for the malware classification task.

A future line of research is the exploration of encryption and compression techniques, and the investigation of their effects in ML classifiers.

Acknowledgements

This research has been partially funded by the Spanish MICINN Projects TIN2015-71799-C2-2-P, ENE2015-64117-C5-1-R, PID2019-111544GB-C22, and supported by

the University of Lleida.

Conflicts of Interest

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

References

- 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. URL: <http://tensorflow.org/>. software available from tensorflow.org.
- Amro, S.A., Alkhalfah, A., 2015. A comparative study of virus detection techniques. International Journal of Computer and Information Engineering 9, 1559 – 1566. URL: <https://publications.waset.org/vol/102>.
- Anderson, H.S., Kharkar, A., Filar, B., Evans, D., Roth, P., 2018. Learning to evade static PE machine learning malware models via reinforcement learning. CoRR abs/1801.08917. URL: <http://arxiv.org/abs/1801.08917>, arXiv:1801.08917.
- Bellman, R.E., 2015. Adaptive control processes: a guided tour. Princeton university press.
- Chen, L., 2009. Curse of Dimensionality. Springer US, Boston, MA. pp. 545–546. doi:10.1007/978-0-387-39940-9_133.
- Dauphin, Y.N., Fan, A., Auli, M., Grangier, D., 2017. Language modeling with gated convolutional networks, in: Proceedings of the 34th International Conference on Machine Learning - Volume 70, JMLR.org. p. 933–941.
- Demetrio, L., Biggio, B., Lagorio, G., Roli, F., Armando, A., 2019. Explaining vulnerabilities of deep learning to adversarial malware binaries, in: 3rd Italian Conference on Cyber Security, ITASEC 2019, CEUR Workshop Proceedings. CEUR Workshop Proceedings, Pisa, Italy. URL: <https://arxiv.org/abs/1901.03583>.
- 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., 2020a. Orthrus: A bimodal learning architecture for malware classification, in: Proceedings of the International Joint Conference on Neural Networks (IJCNN) 2020, Glasgow (UK), IEEE.
- Gibert, D., Mateu, C., Planes, J., 2020b. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. Journal of Network and Computer Applications 153, 102526. URL: <http://www.sciencedirect.com/science/article/pii/S1084804519303868>, doi:<https://doi.org/10.1016/j.jnca.2019.102526>.
- 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.
- Hu, W., Tan, Y., 2017. Generating adversarial malware examples for black-box attacks based on GAN. CoRR abs/1702.05983. URL: <http://arxiv.org/abs/1702.05983>, arXiv:1702.05983.
- Hu, X., Shin, K.G., Bhatkar, S., Griffin, K., 2013. Mutantx-s: Scalable malware clustering based on static features, in: Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13), USENIX, San Jose, CA. pp. 187–198. URL: <https://www.usenix.org/conference/atc13/technical-sessions/presentation/hu>.
- Kolosnjaji, B., Demontis, A., Biggio, B., Maiorca, D., Giacinto, G., Eckert, C., Roli, F., 2018. Adversarial malware binaries: Evading deep learning for malware detection in executables, in: 26th European Signal Processing Conference, EUSIPCO 2018, Roma, Italy, September

- 3-7, 2018, IEEE. pp. 533–537. URL: <https://doi.org/10.23919/EUSIPCO.2018.8553214>, doi:10.23919/EUSIPCO.2018.8553214.
- Krčál, M., Švec, O., Bálek, M., Jašek, O., 2018. Deep convolutional malware classifiers can learn from raw executables and labels only. URL: <https://openreview.net/pdf?id=HkHrmM1PM>.
- Mcgraw, G., Bonett, R., Figueroa, H., Shepardson, V., 2019. Security engineering for machine learning. Computer 52, 54–57.
- Pitropakis, N., Panaousis, E., Giannetsos, T., Anastasiadis, E., Loukas, G., 2019. A taxonomy and survey of attacks against machine learning. Computer Science Review 34, 100199. URL: <http://www.sciencedirect.com/science/article/pii/S1574013718303289>, doi:<https://doi.org/10.1016/j.cosrev.2019.100199>.
- Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., Nicholas, C.K., 2018a. 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.
- Raff, E., Nicholas, C., 2018. Hash-grams: Faster n-gram features for classification and malware detection, in: Proceedings of the ACM Symposium on Document Engineering 2018, Association for Computing Machinery, New York, NY, USA. URL: <https://doi.org/10.1145/3209280.3229085>, doi:10.1145/3209280.3229085.
- Raff, E., Zak, R., Cox, R., Sylvester, J., Yacci, P., Ward, R., Tracy, A., McLean, M., Nicholas, C., 2018b. An investigation of byte n-gram features for malware classification. Journal of Computer Virology and Hacking Techniques 14, 1–20. URL: <https://doi.org/10.1007/s11416-016-0283-1>, doi:10.1007/s11416-016-0283-1.
- Raff, E., Zak, R., Cox, R., Sylvester, J., Yacci, P., Ward, R., Tracy, A., McLean, M., Nicholas, C., 2018c. An investigation of byte n-gram features for malware classification. Journal of Computer Virology and Hacking Techniques 14, 1–20. URL: <https://doi.org/10.1007/s11416-016-0283-1>, doi:10.1007/s11416-016-0283-1.
- Ronen, R., Radu, M., Feuerstein, C., Yom-Tov, E., Ahmadi, M., 2018. Microsoft malware classification challenge. CoRR abs/1802.10135. URL: <http://arxiv.org/abs/1802.10135>, arXiv:1802.10135.
- 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. URL: <http://www.sciencedirect.com/science/article/pii/S0020025511004336>, doi:<https://doi.org/10.1016/j.ins.2011.08.020>. data Mining for Information Security.
- Souri, A., Hosseini, R., 2018. A state-of-the-art survey of malware detection approaches using data mining techniques. Human-centric Computing and Information Sciences 8, 3. URL: <https://doi.org/10.1186/s13673-018-0125-x>, doi:10.1186/s13673-018-0125-x.
- Suciù, O., Coull, S.E., Johns, J., 2019. Exploring adversarial examples in malware detection, in: 2019 IEEE Security and Privacy Workshops, SP Workshops 2019, San Francisco, CA, USA, May 19–23, 2019, IEEE. pp. 8–14. URL: <https://doi.org/10.1109/SPW.2019.00015>, doi:10.1109/SPW.2019.00015.
- Ször, P., Ferrie, P., 2001. Hunting for metamorphic, in: In Virus Bulletin Conference, pp. 123–144.
- Ucci, D., Aniello, L., Baldoni, R., 2019. Survey of machine learning techniques for malware analysis. Computers & Security 81, 123 – 147. URL: <http://www.sciencedirect.com/science/article/pii/S0167404818303808>, doi:<https://doi.org/10.1016/j.cose.2018.11.001>.
- Vergara, J.R., Estévez, P.A., 2014. A review of feature selection methods based on mutual information. Neural Computing and Applications 24, 175–186. URL: <https://doi.org/10.1007/s00521-013-1368-0>, doi:10.1007/s00521-013-1368-0.
- Yu, F., Koltun, V., 2016. Multi-scale context aggregation by dilated convolutions. CoRR abs/1511.07122.
- Zhang, F., Zhao, T., 2017. Malware detection and classification based on n-grams attribute similarity, in: 2017 IEEE International Conference on Computational Science and Engineering, CSE 2017, and IEEE International Conference on Embedded and Ubiquitous Computing, EUC 2017, Guangzhou, China, July 21–24, 2017, Volume 1, IEEE Computer Society. pp. 793–796. URL: <https://doi.org/10.1109/CSE-EUC.2017.157>, doi:10.

1109/CSE-EUC.2017.157.
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.

Appendix A. Neural Network Architectures.

Appendix A.1. Shallow CNN (Gibert et al., 2017).

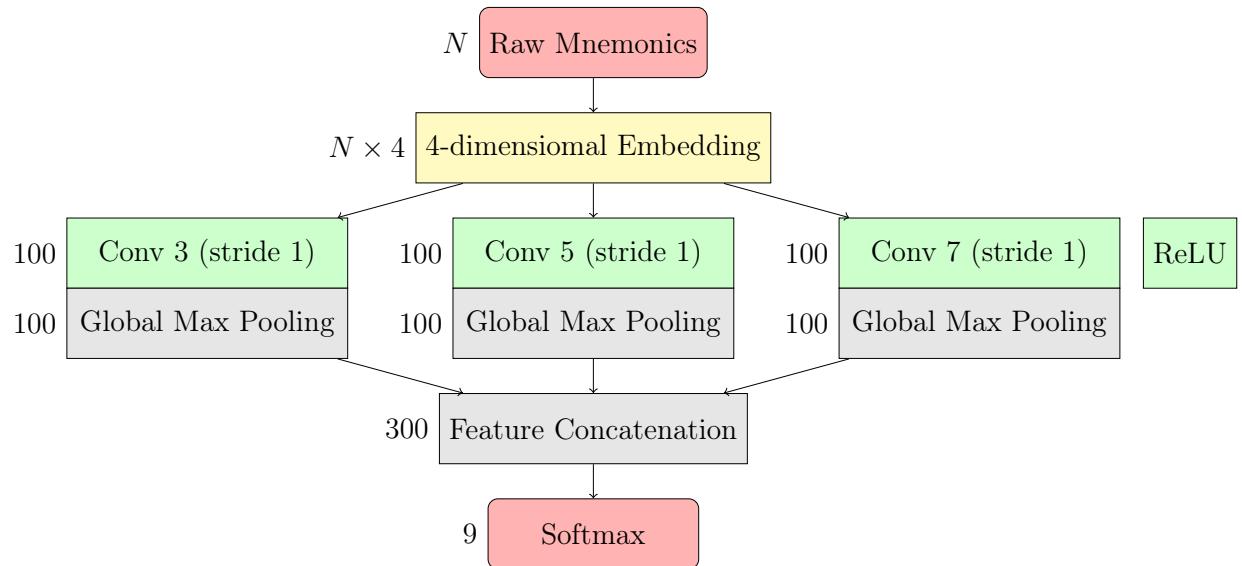


Figure A.18: Opcode-based shallow convolutional neural network (Gibert et al., 2017).

Appendix A.2. MalConv (Raff et al., 2018a).

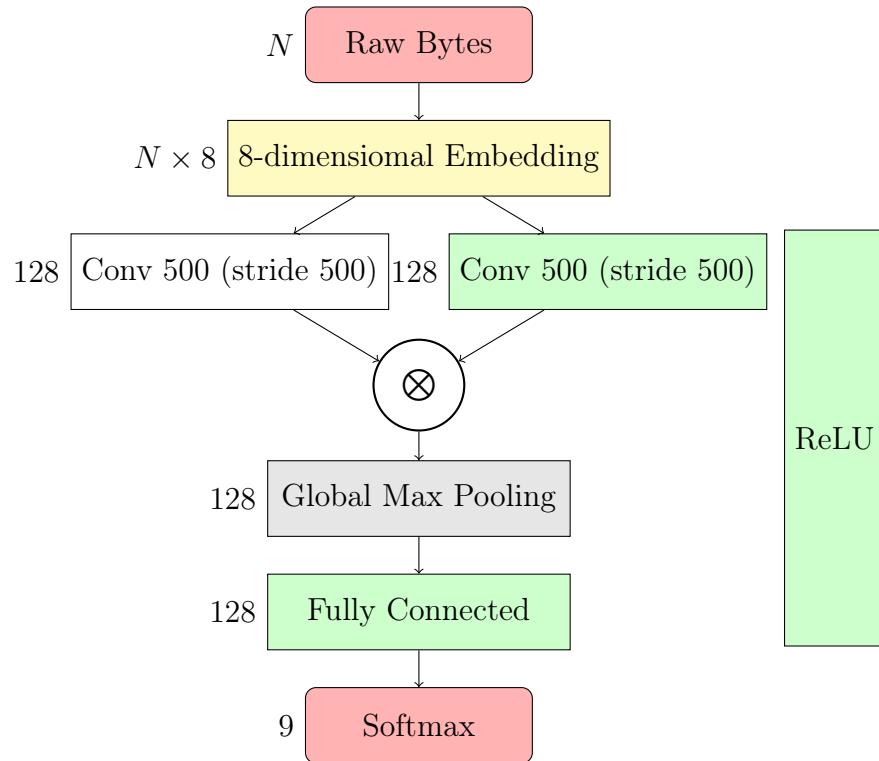


Figure A.19: MalConv architecture (Raff et al., 2018a).

Appendix A.3. DeepConv (Krcál et al., 2018).

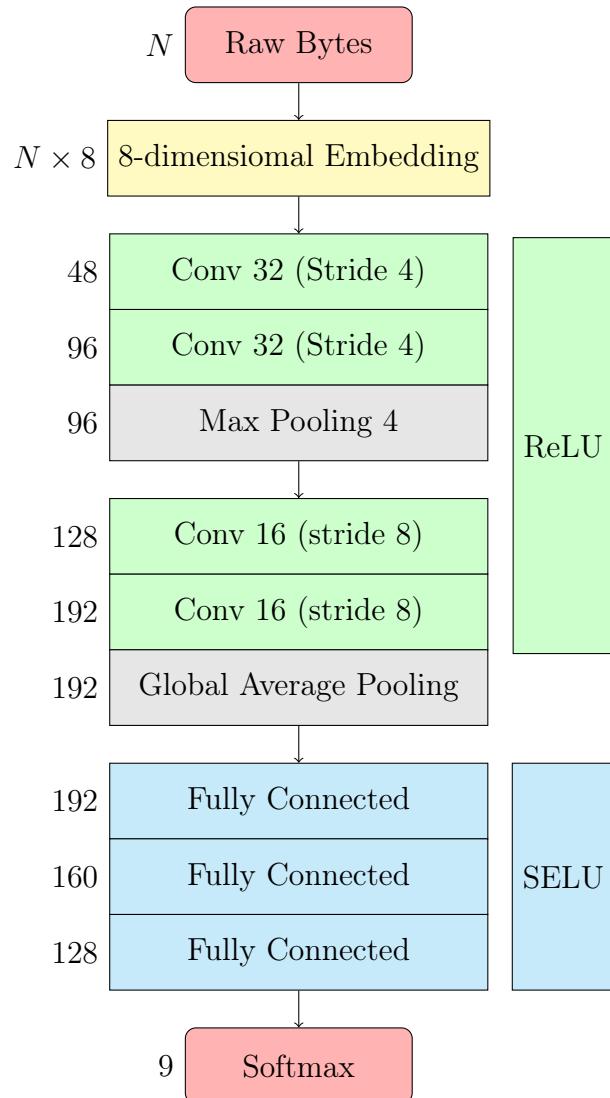


Figure A.20: DeepConv architecture (Krcál et al., 2018).

Appendix A.4. Structural entropy CNN and Multiresolution CNN (Gibert et al., 2018)

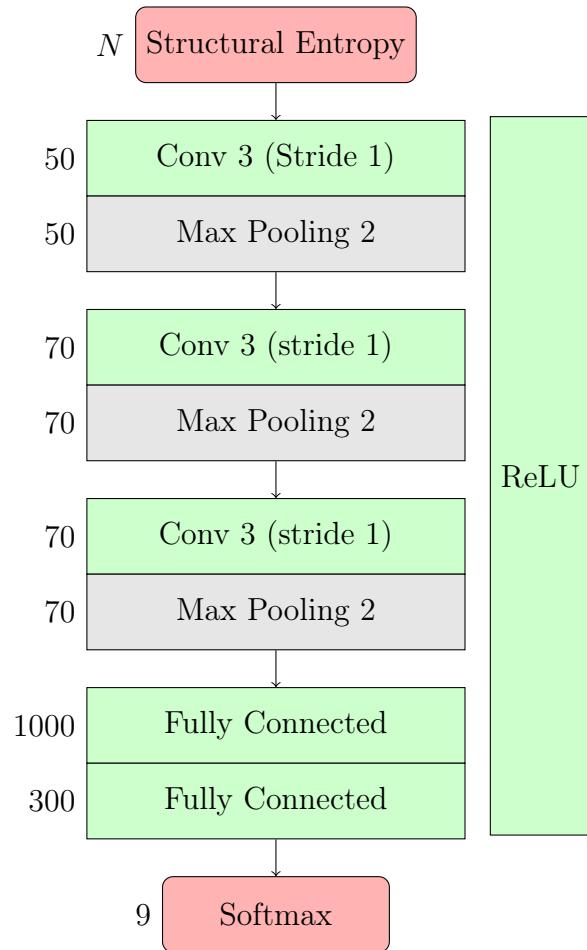


Figure A.21: Structural Entropy CNN (Gibert et al., 2018).

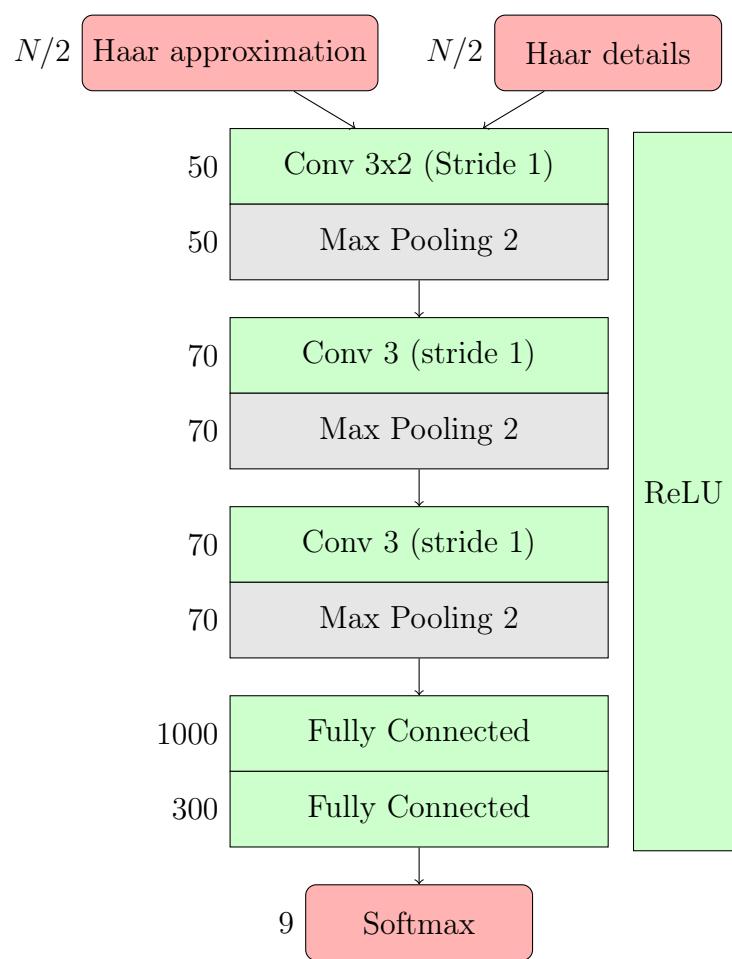


Figure A.22: Multiresolution CNN (Gibert et al., 2018).