Detection of metamorphic malware packers using multilayered LSTM networks

Erik Bergenholtz[1], Emiliano Casalicchio[1,2], Dragos Ilie[1], and Andrew Moss[1]

[1] Blekinge Institute of Technology {ebz,awm,dil,emc}@bth.se
[2] Sapienza University of Rome, Italy emiliano.casalicchio@uniroma1.it

Abstract. Malware authors do their best to conceal their malicious software to increase its probability of spreading and to slow down analysis. One method used to conceal malware is packing, in which the original malware is completely hidden through compression or encryption, only to be reconstructed at run-time. In addition, packers can be metamorphic, meaning that the output of the packer will never be exactly the same, even if the same file is packed again. As the use of known off-the-shelf malware packers is declining, it is becoming increasingly more important to implement methods of detecting packed executables without having any known samples of a given packer. In this study, we evaluate the use of recurrent neural networks as a means to classify whether or not a file is packed by a metamorphic packer. We show that even with quite simple networks, it is possible to correctly distinguish packed executables from non-packed executables with an accuracy of up to 89.36% when trained on a single packer, even for samples packed by previously unseen packers. Training the network on more packer raises this number to up to 99.69%.

Keywords: packing · packer detection · security · static analysis · machine learning · deep learning

1 Introduction

There is a constant arms race going on between malware authors and malware analysts. As anti-malware tools get better at detecting malware, the malware authors are being forced to adapt new strategies to hide their malware. Modern anti-malware tools rely mainly on two approaches: signature-based detection, and detection based on heuristics. The first method detects malware by searching for exactly matching unique byte-strings, called *signatures*, within an analyzed file, while approaches based on heuristics estimates the behavior of the analyzed code by e.g. enumerating called functions. Since both of these methods rely on analyzing the malware itself, a common way of avoiding detection is to hide the malicious software using tools called *packers*, which completely hide the original code from analysis. It is reported that up to 92% of all malware is hidden

The final authenticated version is available online at https://doi.org/10.1007/978-3-030-61078-4_3

this way [12], and 35% of these are hidden using custom, previously unseen packers [23]. This large number of unknown packers, combined with the inability to analyze the malicious code itself, results in anti-malware tools having a harder time detecting potential malware. To make matters worse, some packers are designed to procedurally generate packed executables that always look different, even if the original file is the same.

Detection of packed binaries, rather than malicious code itself, is useful in several ways. As mentioned, the malware itself cannot be detected using conventional methods when it is hidden. However once a file is determined to be obfuscated it can be flagged as high priority for further analysis. If such detection is used in e.g. a network intrusion detection system (NIDS), the search space to identify the responsible file of an intrusion could be decreased. Such files could also be stopped from entering the network until they are checked and cleared by an administrator. Since a large portion of the tools used to hide malware are custom made, and therefor not previously known, studying ways to generically detect these kinds of obfuscation techniques is important. Despite this, studies conducted on packer detection (discussed in Section 8) do generally not evaluate how general the proposed methods are, but they only evaluate on packers known by the model. A notable exception is a study by Bat-Erdene et al from 2017 [9].

The purpose of this study is to determine whether deep learning, in particular *recurrent neural networks*, can be used to differentiate between the procedurally generated code mentioned above, and compiler generated code. We will determine whether or not this is possible by training a neural network on several data sets derived from a large set of off-the-shelf packers, and evaluating how general these models are. These experiments are described in Section 6. Our results show that neural networks can be trained to make the distinction, not only for packers in the training set, but also for previously unseen packers. To the best of our knowledge, we are the first to use deep learning to solve this problem, and we have evaluated our approach on the largest set of packers in the literature. Our two main contributions are *a)* showing that deep learning can be used to train models capable of distinguishing between obfuscated code and compiler generated code in the general case, and *b)* our classification of a very large amount of run-time packers.

The rest of this paper is structured as follows. In Section 2, the concepts of a *packer*, *metamorphic packer* and *polymorphic packer* are defined. Section 3 describes the recurrent neural network used for packer detection in this study, and Section 4 discusses how data generation and processing was performed. Section 5 describes how the set of packers studied in this paper was selected. The experiment design is explained in Section 6. Results are shown in Section 7. Previous work in the area of packer detection is laid out in Section 8, and finally conclusions and future work are presented in Section 9.

2 Background

The tools used by malware authors to hide their malware, referred to as *packers*, have evolved from earlier tools that produced self-extracting archives [27]. The term "packer" originally referred to a program that packed a set of files into a single package. This meaning has shifted over time to refer to tools that transform executable files into another form that can reproduce the original at run-time. This drift in terminology has led to competing definitions amongst the work in this area. For clarity, we pin down definitions of these commonly used terms. We also discuss the operation of a packer, as well as different kinds of packers.

2.1 Terminology

Throughout this paper, we will use the following terms to talk about packers, and the concepts surrounding them:

- A *packer* is a program that transforms an executable into two parts: an unpacking stub and the data that it operates upon.
- An *unpacker* or *unpacking stub* is a piece of code that converts data into code.
- The *original program* is an executable whose signature is being hidden by the packing process.
- The *packed data* is a binary stream from which the original program can be reconstructed by the unpacker.

Most packers will perform (up to two) transformations when creating the packed data: compressing the data, and/or encrypting the data. Typically packers that compress the packed data are referred to as *compressors*, and packers that encrypt the packed data are referred as as *crypters* [30]. These transformations are not mutually exclusive and it is possible for a packer to be both a compressor and a crypter.

When checking for malware using *signature-based* detection, the stream of bytes in the unpacker and packed data are compared to known samples (typically by comparing hashes). In order to avoid detection, a *polymorphic packer* will create a different unpacker and packed data stream on each executable. This may be achieved, for example, by encrypting the code via a different key on each execution of the packer [27]. In a similar fashion, a *metamorphic packer* will avoid detection by generating unpackers where the code is semantically equivalent but not identical [27]. Programming using macros instead of actual code, where each macro represents a set of different representations of the same operation, can be used to accomplish this [15]. A *monomorphic packer* will produce the same unpacker and packed data stream for each execution on the same input original program [28].

2.2 Typical Operation

Packers operate on executable files, which can be either an original program, or an executable that has already been processed by one or more packers. This should not matter, since the original executable is simply data to the packer. The executable is transformed in some way, commonly through compression or encryption, to hide the original code. Following this, an unpacking stub is created and bundled with the transformed executable in a new executable file. The entry point of this executable file points to the start of the unpacking stub, which will inflate or decrypt the original executable into memory at run-time. Typically this is done in one single pass, unpacking the whole original into memory at once, however there are advanced packers that use multiple passes [28].

Once the original code is unpacked into memory, the unpacking stub will hand over execution to the unpacked application. This is typically done through a *tail jump* to the *original entry point* (OEP). The tail jump is commonly obfuscated, e.g. by pushing the OEP to the stack and "returning" to it, to hide where in memory the original code starts.

It is common for packers to employ techniques for making the unpacking stub itself harder to analyze as well. Common techniques include embedding random data in the code, loading libraries at run-time, overlapping instructions, as well as poly- and metamorphic code. Code can be metamorphic either by procedurally generating the assembly code itself by choosing between synonymous assembly sequences, or by inserting dummy basic blocks into the control flow graph of the program. `Morphine v.2.7` [17] uses both of these techniques, and also inserts junk code, i.e. code that is semantically identical to a `NOP`, into the unpacking stub to make analysis harder [14].

3 Neural Network Design

The data that is being analyzed in this study, discussed at length in Section 4.1, is a sequence of x86 assembly instructions. Since these sequences are slices of real code, context is crucial. For instance, while the operation of an `XOR` opcode will always be the same in any given executable, the purpose can vary widely depending on how it is used. In some contexts an `XOR` opcode might be used to efficiently clear a register, while in another context it could be used to decrypt packed data that was encrypted with an XOR-cipher. Because of this, the general design of the neural network evaluated in this study is a recurrent neural network, as they are well suited for learning a context sensitive sequence of data.

In particular, the neural network used in this study is made up of a multi-layered LSTM network, and a fully connected binary classifier. The multi-layered LSTM network has two layers, each with 128 nodes. The second of these two layers feed into a dropout layer with 50% dropout to mitigate overfitting. The dropout layer feed into the binary classifier, which has three layers of 128, 64 and 1 nodes respectively.

The first two layers of the fully connected binary classifier use a $ReLU$ (Rectified Linear Unit) activation function, while the last layer uses a sigmoid function.

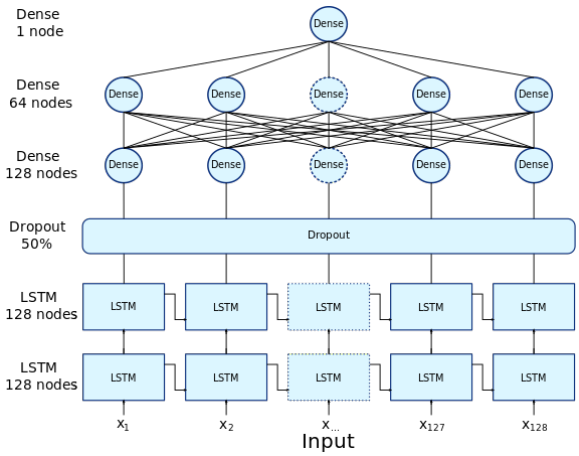

Fig. 1. Neural network design used in this study.

ReLU is used as it makes training the network easier while still yielding good results, while the sigmoid function is used to make sure the output of the network is a probability between 0 and 1. We chose a sigmoid function, as opposed to a softmax function, since we are performing binary classification in this study. The network is illustrated in Figure 1.

The input of the network consists of the first 128 instructions of each sample file. A sliding window is used to specify each time-step in the sequence, where the window size $w = 1$ is used. Both the number of instructions and the size of the sliding window were determined through a process that is described in [10]. Each input is labeled $l \in \{0, 1\}$ where 0 means the instructions come from a non-packed executable, and 1 means they come from a packed executable. The encoding of the x86 instructions is discussed in Section 4.1.

3.1 Training

The neural network was trained for 50 epochs, with a batch size of $10*n$, where n is the number of packers included in training. A small batch size means reduced training time and memory requirements, while letting the network update its weights a large number of times to facilitate learning. Each batch consisted of both packed and non-packed samples in equal amounts, to ensure that the neural network had equal exposure to positive (packed) and negative (non-packed) samples. Because each sample is only seen by the network exactly once per epoch, the network need to be trained for multiple epochs to allow the network to make a sufficient number of updates to the weights of each node, which is why the network is trained for 50 epochs. The model with the best accuracy was saved and used for the evaluation.

An epoch is a pass over all training and validation data exactly once.

4 Data Collection and Preprocessing

In this section the collection and generation of the raw data used in this study is laid out, as well as the preprocessing and filtering steps that were taken to construct the final data set.

4.1 Data Encoding

The goal of this study is to determine if it is possible to utilize neural networks to differentiate metamorphic code from compiler generated code. As such, the raw data used will be binary code, which is structurally complex. It is therefore necessary to find a way to represent this data, so the neural network can understand it.

The first issue to address was which data to feed to the network. A trivial approach would be to use the raw byte values of the code, but the issue with this approach is that some operations that are semantically different share bytes. For instance, both `JMP` and `INC` start with the byte `0xFF`. Extending this to the whole opcode does not solve this problem, as `JMP [EAX]` and `INC [EAX]` are semantically different but have the opcodes `0xFF2` and `0xFF0` respectively, which are numerically very close to each other. Using the whole numerical values for an instruction and its arguments has the opposite problem; `JMP EAX` and `JMP 0x1234` are semantically similar, but their numerical values, `0xFFE0` and `0xe92f120000`, are very different.

The encoding scheme that we decided on was to map the assembly mnemonics of the x86 instruction set to a list sorted according to the order given in chapter 6 of the x86 manual [4], where the mnemonics are grouped by the type of operation they perform (e.g. moves, jumps or arithmetic). Mnemonics not described there were sorted alphabetically at the end of our list. The data fed to the neural network are the indices of these mnemonics, meaning that semantically similar operations will have similar indices, thus solving the issues mentioned above. Based on an evaluation detailed in [10], we chose to only consider the first $n = 128$ instructions after the entry point of each executable file. This results in fast execution, while still retaining a good average accuracy. Disassemblies that were shorter than n instructions were padded with meaningless values. However if the last disassembled instruction was a direct jump it was followed, and disassembly continued from there. Because the disassembly will only be too short if it reaches the end of an executable section, we know that this has to be unconditional control flow (either `JMP` or `RET`), as otherwise execution would risk "falling outside" the code. Therefore we only need to consider the last instruction. Files that could not be disassembled at all, or were not recognized as Portable Executable (PE) files, were excluded from the data set.

`https://gist.github.com/erikbergenholtz/a653d46db64c2ce490af91698f75e992`

4.2 Collection and Generation of Raw Data

The basis for the data set used in this study were 1904 executable files retrieved from the `C:/Windows` directory of our reference system. These files were packed once by each of the 42 packers found in the prestudy in Section 5 (positive samples), and were also included in the data set in their original form (negative samples). Many packers failed to pack the whole set of 1904 files. On average, each packer could pack 1358 files. Since some of our experiments include multiple packers in the training set, we augmented the negative samples with the 13002 DLL files found in the `C:/Windows` directory of reference system. All of these files were preprocessed according to Section 4.1, resulting in a total of 61535 positive samples and 12549 negative samples after preprocessing, meaning that the full test set consisted of 74084 files.

These 74084 files were split into one training set, one validation set, and one test set. The validation and test sets each consist of 10% of the total amount of data each, i.e. 7426 files, and the remaining 59232 files are in the training set.

It is important to note that while the full data set is unbalanced, with almost five times more positive samples than negative samples, the three subsets mentioned above are always balanced when used. If, for example, the neural network is trained on a packer with 1000 samples, then 1000 negative samples are used. If a set of packers with a total of 13000 samples is used for training, then this set is capped at 12549 samples, and an equal amount of files is used from each packer. In other words, in both training, validation and test sets there is *always* balanced data, despite the fact that the data set as a whole is unbalanced.

5 Packer Prestudy

A prestudy was conducted to determine which packers to include in the main study. We had two criterion for including a given packer in the main study: availability and relevance. We consider a packer to be available if and only if we can legally acquire a copy of it without purchasing it. This means that commercial tools are out of the scope of this study, unless they provide a free demo or trial version. A packer is considered to be relevant if and only if it is metamorphic, possible to execute on a modern operating system, and is able to pack 32-bit PE files. We chose to not make the ability to pack 64-bit PE files a requirement, as a lot of the packers we found in the prestudy were 32-bit applications, and we wanted to include as many packers as possible. We still consider 32-bit packers relevant, as they can run on modern 64-bit systems.

A total of 180 packers, listed in Table 1, were identified and considered for this this study. The available packers were evaluated for relevance on a Windows 10 virtual machine, by packing the same executable twice with each packer. The two resulting executables were disassembled with `objdump`, and these disassem-

Windows 10 Education 32-bit, build 17763.316
Windows 10 Education 32bit, build 17763.316
`objdump -d <FILENAME>`

Table 1. Packers included in the selection process. Packers marked with green were included in the study, and red were unavailable. Yellow are metamorphic, but not included.

AASE	Aegis Crypter	AHT Entry Point Protector	AKALA v3.20
Alex Protector	Allaple	Alloy v4.3.21.2005	Alternate_exe v2.220
AntiCrack protector	AntiCrack protector pro	AntiUCPE v1.02	APack v0.98
Armadillo	ARMProtector v0.3	ASPack v2.43	ASprotect v2018.3
ASprotect v2.78	Beria 0.07	Berio	BeRoEXEPacker
BJFNT v1.3	CelsiusCrypt	CodeCrypt v0.164	Code Virtualizer v2.2.1.0
ComCryptor v1.80	Corso v5.2	Crinkler 2.1a	Crunch v1.0
CRYPToCRACk's PE Protector	CryptoLock v2.0	Daemon Crypt 2.0	DalKrypt v1.0
Diet	DingBoy PE-Lock v1.5	DragonArmor v0.4.1	Drony Application Protect v3.0
Enigma v6.00	!EPack v1.0	!EPack v1.4	EPProtect
Escargot	Excalibur v1.03	exe32pack	EXECryptor v1.3
EXEFog v1.12	EXEJoiner	EXEPack	ExeSax v0.9.1
ExeStealth	eXPressor	FileXPack	FSG v1.3
FSG v2.0	GHF Protector	HidePX v1.4	Hmimy's Protector
HuffComp v1.3	Hyperion	JDPack	KillFlower v1.2
KKrunchy v0.23a2	KKrunchy v0.23a	KKryptor	Krypton
LameCrypt	LiteProtect	LZEXE v0.91	LZEXE v0.91e
MarCrypt v0.1	marioPACKer v0.0.3	MaskPE	Masspecer v3.0
Mew v11	MicroJoiner	Molebox	Morphine 1.5
Morphine 1.6	Morphine 1.7	Morphine 1.9	Morphine 2.7
Morphine 3.5	Morphnah	MPRESS v1.27	MPRESS v2.18
MPRESS v2.19	[MSLRH]	Mucki's Protector v1.0	MZOoPE v1.0.6b
NakedPacker v1.0	NeoLite v2.0	NFO v1.0	NiceProtect
NoobyProtect	NoodleCrypt v2.0	nPack	NSAnti (Anti007)
NsPack v3.7	NTKrnl	Obsidium v1.6.6	Obsidium v1.6.7
ORiEN	PackerFuck	PackMan v1.0	Pack v1.0
PCGuard v6.00.0540	PCShrink v0.71	PE-Armor	PEBundle
PECompact	PECRP v1.02	PECrypt32 v1.02	PEDiminisher
PELockTide v1.0	PELock v2.08	PE.ncrypt v3.0	PE.ncrypt v4.0
PenguinCrypt	PE Ninja	PEPaCK	PE.Prot
PersonalPrivatePacker	PEShiELD v0.25	PEShrinker	PESpin
PEstil	PETITE v2.4	PeX	PKLITE32
Pohernah v1.1.0	PolyCrypter	PolyCrypt PE	PolyEnE v0.01+
Private EXE Protector v2.0	RCryptor	RDG Tejon Crypter	ResCrypt
RJoiner	RkManager11	RLPack 1.21	RPolycrypt
ScrambleUPX v1.07	SecureCode	Sentry	ShareGuard v4.0
Shrinker v3.4 demen	Shellter v7.1	SimplePack v1.0	SimplePack v1.3
SLVc0deProtector v1.12	STonePE	tELock	Themida v2.4.5.0
ThinApp	Trap v1.21	UCFPE v1.13	Unk0wn Crypter v1.0
Unopix v0.94	Unopix v1.10	Upack	UPolyX
UPX v3.91w	UPX v3.95	USSR v0.31	VBox
VGCrypt v0.75	VMProtect v3.3	VPacker v0.02.10	WinKript
Winlicence v2.4.5.0	Winlite	WinUpack	WWPack32 v1.12
WWPack32 v1.20	XCR v0.13	XProtector	XXPack v0.1
YodaCrypter v1.3	YodaProtector v1.03	ZCode v1.0.1	ZProtect

blies were compared. A packer is considered to be metamorphic, and therefore relevant, if the disassemblies differ.

We chose to compare the disassembled code for differences, rather than the files themselves, as we are only interested in packers where the unpacking stub is generated dynamically. Cases where different encryption keys are used each time, i.e. polymorphic packers, are not of interest in this study.

Monomorphic packers are excluded from the study as they are trivial to detect using signature-based detection, and because they would only provide a single data point for the neural network to learn from.

6 Experiments

Two experiments were performed in this study, both of which are laid out in this section. We also discuss how the disassembly engine used in these experiments was selected.

6.1 Choice of Disassembler

Since the neural network works with mappings of opcode mnemonics, it is essential how to extract the mnemonics. We considered two well known, off-the-shelf disassemblers for this study: `objdump` [1] and `radare2` [2]. These two tools operate differently, in that `objdump` is a *linear* disassembler while `radare2` is a *recursive* disassembler. This means that `objdump` will disassemble a program from the first instruction to the last in the order instructions are laid out in the file, while `radare2` will disassemble one block of code at a time, following jump instructions along the way. The consequences of this are that `objdump` will be able to disassemble all code in the file, but may also disassemble embedded data by mistake. `radare2`, on the other hand, won't disassemble any data, but may end up seeing very small portions of the code if it encounters indirect jumps. The tools were evaluated on a subset of the data set used in this study, and from this evaluation it was clear that `objdump` is in general able to disassemble larger parts of the files than `radare2`. For this reason, we chose to use `objdump` in this study. More details on the evaluation can be found in [10].

6.2 Experiment Design

The experiments laid out below were all performed with the parameters and procedures described above. The results can be found in Section 7.

Training on a Single Packer In the first experiment, we trained the neural network on a single packer at a time for each packer included in the study. This allows us to determine which packers produce the model that can most accurately distinguish packer generated code from compiler generated code, even for packers that the neural network has not been exposed to. Being able to train such a general model with samples from a single packer would be very beneficial, as it would allow us to detect unknown packers, even with a small training set.

Training on $n - 1$ Packers, Evaluate on Excluded One In the second experiment, a model was trained on all packers included in the experiment except for one. The model was then evaluated on the excluded packer. This was done for the ten packer families that yielded the most accurate models in the first experiment. This experiment represents a realistic scenario in which we have access to samples of many, but not all, packers, and where a new unseen packer is being scanned by the anti-malware tool. As with the previous experiment, begin able to train such a general model would be highly beneficial, as it would allow us to detect unseen packers. Since it will be exposed to more kinds of metamorphic code, it is our hypothesis that the accuracy of these models will be higher than that of the first experiments.

`objdump -Mintel -D --start-address <ENTRY POINT>`

6.3 Evaluation

The performance of each model was evaluated by estimating the probability of a given file being packed (p_{packed}) once for each file in the unseen test set of each experiment. Since we are interested in a binary prediction (packed or not packed) we applied Equation 1 to determine whether or not a file was considered to be packed or not.

$$prediction = \begin{cases} packed & \text{if } p_{packed} > 0.5 \\ non\text{-}packed & \text{if } p_{packed} \leq 0.5 \end{cases} \tag{1}$$

Two different test sets were created for each of the two experiments described above. For the first experiment, the first test set consisted of executables packed by the packer used for training. The second test set consisted of packed files from all packers. For the second set of experiments, the first test set consisted of files packed by the one packer that was excluded from the training set. The second test set consisted of all packers included in the experiment. All test sets also contain non-packed files in the same amount as packed files.

Of these four test sets, we are mostly interested in the evaluation on all packers for experiment one, and in the evaluation of the excluded packer for experiment two. This is because these two evaluations are performed on packers that are *not* included in the training sets of the experiments, and will therefore show how general the trained model is. A more general model will be able to more accurately detect unseen packers, which is highly desirable.

Since all our data is labeled, we are recording the number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) for the two evaluations of each model. A true positive in our case is a file that has been classified as packer and in fact is packed, and a true negative is a non-packed file classified as such. Using these values, we can calculate accuracy, precision and recall for all models.

7 Results

The results of the experiments described in Section 6 are laid out below.

7.1 Model Trained on a Single Packer

The accuracy of each model trained on a single packer can be seen in Figure 2. For each model, the accuracy of evaluating the models on the packer itself, as well as all packers in bulk, is shown. From the figure it is clear that most models work well when classifying files packed by the packer used for training. However, we are more interested in seeing how well the models generalize onto the unknown packers. Here, some of the models perform well, with the best being the model trained on `EXEFog v1.12` with and accuracy of 89.36%, and the worst being `PE Ninja` with an accuracy of only 51.16%.

These results are very promising, as it means that by training an RNN on only samples packed by `EXEFog v1.12`, we can get a model that can correctly distinguish files from any of the 42 packers in 89.36% of the cases. This, combined with the recall of 81.75% and a precision of 96.43%, as seen in Figure 4 in the Appendix, makes for a good model for detection of executables packed by a metamorphic packer, even the packer is unknown to the network.

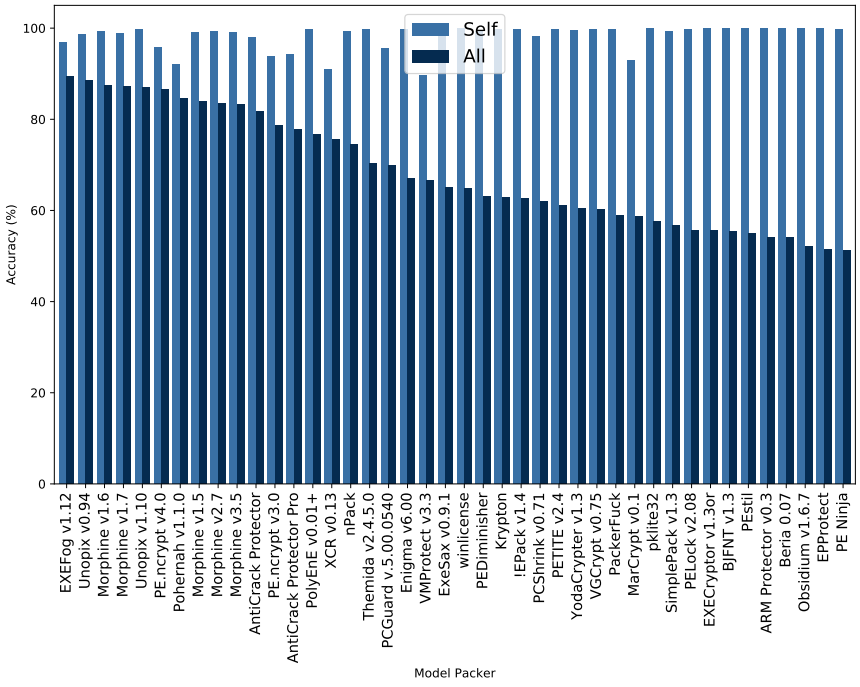

Fig. 2. Accuracy of model trained on the individual packers, when evaluated against only the packer included in the training set (Self), and all packers included in the study (All).

7.2 Model Trained on $n - 1$ Packers

Figure 3 shows the accuracy of the models trained on the ten packers that yielded the most accurate models in the previous experiment, with one packer being excluded from the training set. From the graph, we can tell that the resulting models have a very high accuracy, both when evaluated on the excluded packer and when evaluated on all packers in the training set. The best model was the one trained on all packers but `Themida v2.4.5.0`, where an accuracy of 99.69% was achieved when evaluated on only `Themida v2.4.5.0`, and 97.01% when evaluated on all packers in the training set. The other models show a very high

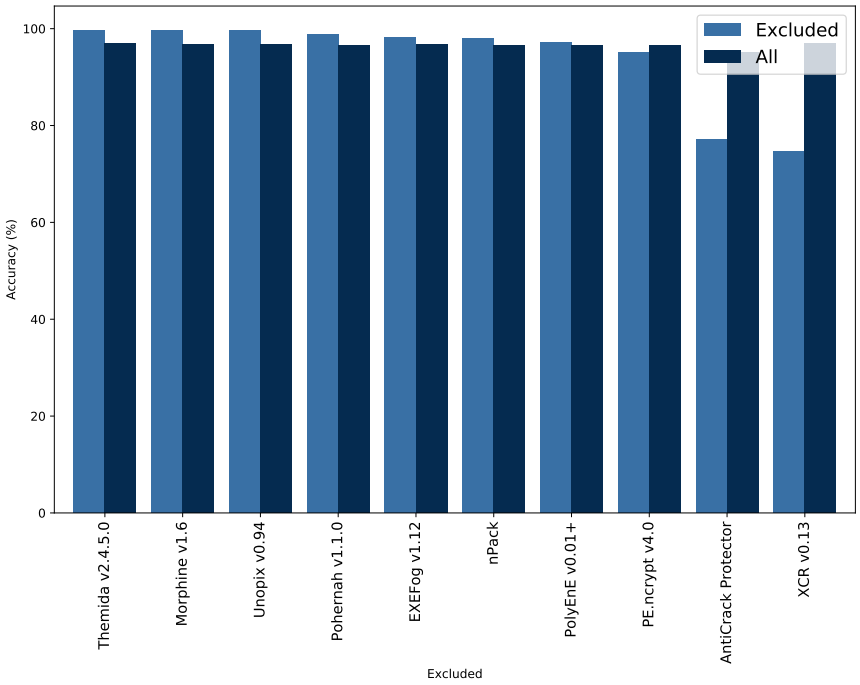

Fig. 3. Accuracy of model trained on N-1 packers, when evaluated against only the packer excluded in the training set (Excluded), and all packers included in the training set (All).

accuracy in general as well, with accuracies above 95% for all packers but two, as shown in the figure. The recall and precision follow the same pattern, as shown in Figure 5 in the Appendix.

Keeping in line with the results from the previous experiment, these results show that it is indeed possible to train a recurrent neural network on a subset of all metamorphic packers, while retaining the ability to accurately distinguish the metamorphic code from compiler generated code of a normal, non-packed executable.

8 Related Work

Although many of the following papers use the term "polymorphic", the packers they study is metamorphic according to our terminology, as described in Section 2.1.

A number of methods have been proposed to address the problem of detection of metamorphic packers, ranging from advanced signature-based detection and entropy analysis to steganalysis.

Signature-based detection is explored by Křoustek et al [21], and Naval et al in [24] and [16]. The approach taken by Křoustek et al is part of the Retargetable Decompiler project [3], and uses handcrafted heuristic signatures. In their study, they demonstrated that the approach can identify metamorphic packers with an accuracy of 98%. Naval et al, on the other hand, use the Smith-Waterman algorithm together with multiple sequence alignment to generate signatures. The method reached an accuracy of 92.5% and 99.0% when evaluated on `ASPack` and `PECompact` respectively [24], and was later extended by parallelizing the Smith-Waterman algorithm, yielding a speed up of up to 49.19 times the original speed, while maintaining accuracy [16].

Ban et al [5,6] used string-kernel-based support vector machines for packer identification, thus bridging the gap between signature-based and machine learning-based detection. Their method could identify which packer was used to pack a certain executable with and accuracy of 91.42%, thus outperforming `PEiD`.

Machine learning approaches of different kinds have also been studied in multiple articles. Hubballi et al evaluated two approaches in 2016 [18]. The first was a semi-supervised approach trained on data from the PE header, with an accuracy of 98.97%, and the second was clustering approach based on the assumption that packers mutate their memory at run time. This method reached an accuracy of 100% for certain packers. Lee et al [22] studied the use of stacked RNNs and Convolutional Neural Networks (CNN) to classify Android malware. Features are extracted using gated recurrent units (GRU), optimized by an additional CNN unit. The method was shown to be robust against obfuscation, and were able to detect 99.9% of the analyzed obfuscated samples. Kancherla et al used Byte and Markov plots to extract features which were used to train an Support Vector Machine [20]. They concluded that the features extracted using the Markov plots performed better, with detection accuracies ranging from 83.94% for `Armadillo` up to 99.05% for `Themida`.

Xie et al proposed the use of a sample-based Extreme Learning Machine (ELM) system for run-time packer detection [29]. Their hypothesis was that the system would be less sensitive to erroneous or missing data if it was sample-based, which was confirmed by experiments in which the proposed system performed better than other ELMs, and reached a detection accuracy of 69.74%.

Bat-Erende et al [7,9,8] studied the use of entropy analysis for packer detection, as did Jeong et al in [19]. In all four studies, the entropy of the executable in memory was calculated while the unpacking stub was running. Using this analysis, Jeong et al could correctly identify the OEP of a packed binary in 72% of their tests [19]. Bat-Erende, meanwhile, could classify files as packed or unpacked with a true positive rate of 98.0%, and an accuracy of 90.4% on files packed once [7], and on average 98.0% on files that were packed multiple times [8]. In [9] they showed that it is also feasible to use this method to detect unknown packers, with an average accuracy of 95.35%. In a similar vein, Sun et al [26] trained statistical classification models on randomness profiles of executables, extracted with a sliding window, to classify packed executables. The

method was shown to have a precision between 95.5% and 100% for certain packers.

Steganalysis, the study of detecting hidden communication inside digital data, was proposed as a means of packer detection by Brugess et al in [13]. Their method converts the executable to a gray-scale image, from which features are then extracted to train a support vector machine. The evaluation of this approach show an accuracy of 99.49%.

More recently, virtual machine (VM)-based obfuscation has been observed in industry-grade obfuscation solutions, such as `VM Protect` and `Themida`, and in advanced malware [25]. When this technique is used, the original machine code (e.g. x86) is converted to a byte code used by the VM. The byte code is based on a instruction set architecture (ISA) chosen randomly at the time of conversion. This makes reverse-engineering very time-consuming. The deobfuscation method presented in [25] relies on static analysis. However, it is not very efficient because it requires more or less full understanding of the VM and needs to be repeated for each obfuscator encountered [11]. On the other hand, [11] proposes a novel method of program synthesis based on Monte Carlo Tree Search (MCTS). Their implementation, called Syntia, allowed them to synthesize with more than 94% success rate the semantics of arithmetical and logical instruction handlers in `VM Protect` and `Themida` obfuscators.

9 Conclusions and Future Work

The results presented in Section 7 show that it is indeed possible to train a recurrent neural network to distinguish between non-packed compiler generated code and the unpacking stub generated by a metamorphic packer. The results also show that it is possible for such models to not only make the distinction for packers included in the training set, but that a high level of accuracy can also be reached for detecting previously unseen packers.

Including a single packer in the training set results in at most an accuracy of 89.36% when the model is evaluated on all packers included in the study. This was achieved by training on `EXEFog v1.12` with a sliding window size of $w = 1$, and the model also had a precision of 96.43% and a recall of 81.75%. These metrics shows that the model performs well, and that this method shows a lot of promise.

Training the RNN on a set of packers and evaluating it on a single excluded packer, reinforces this point. When using a set of ten packers and training on all but one, we achieve an accuracy of 99.69% at most when training on all ten packers except for `Themida v2.4.5.0`. The other packers evaluated this way show generally high performance as well. This shows that as the number of packers included in the test set goes up, its ability to make accurate predictions about unseen packers goes up as well.

As the aim of this study was to simply explore the feasibility of using recurrent neural networks to distinguish between non-packed compiler generated code and metamorphic unpacking stubs, the encoding scheme used to encode

the training and test data is rudimentary and naive. In future studies, we will explore how the encoding affects the accuracy of the trained models. In particular, we will explore whether or not the output of binary analysis methods can be used to extract more meaningful information from the PE files, to improve the accuracy of the network presented in this study.

Appendix Recall and Precision

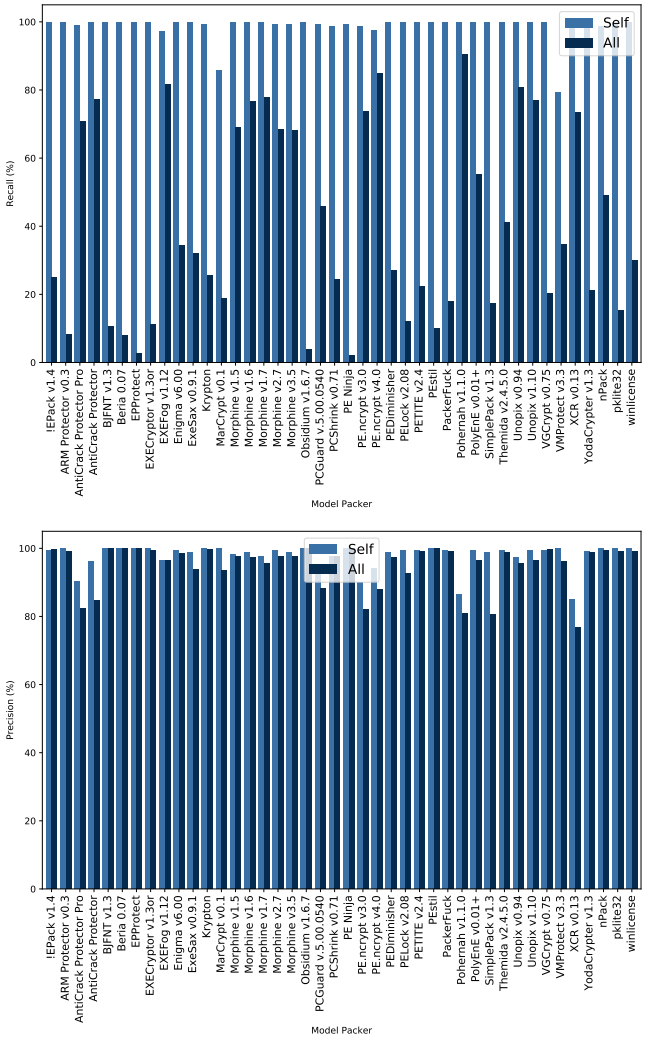

Fig. 4. Recall (top) and precision (bottom) of model trained on the individual packers, when evaluated against only the packer included in the training set (Self), and all packers included in the study (All).

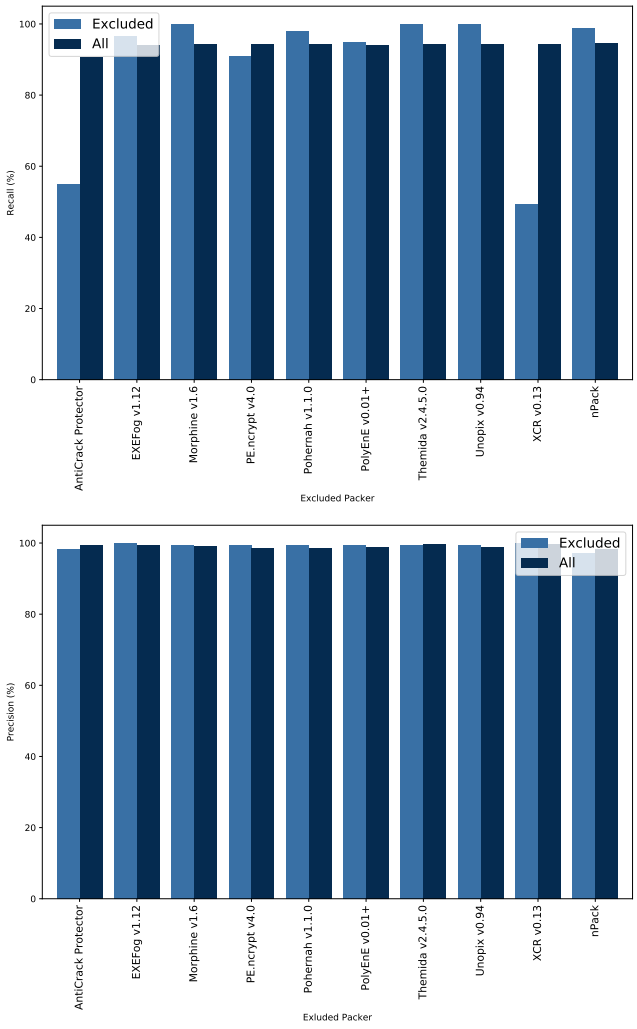

Fig. 5. Recall (top) and precision (bottom) of model trained on N-1 packers, when evaluated against only the packer excluded in the training set (Excluded), and all packers included in the training set (All).

References

1. Objdump. https://sourceware.org/binutils/docs/binutils/objdump.html, accessed: 2020-01-16
2. Radare2. https://www.radare.org/r/, accessed: 2020-01-16
3. Retargetable decompiler. https://retdec.com/, accessed: 2019-05-08
4. Intel® 64 and IA-32 ArchitecturesSoftware Developers Manual (May 2019)
5. Ban, T., Isawa, R., Guo, S., Inoue, D., Nakao, K.: Application of string kernel based support vector machine for malware packer identification. In: The 2013 International Joint Conference on Neural Networks, IJCNN 2013, Dallas, TX, USA, August 4-9, 2013. pp. 1–8. IEEE (2013), https://doi.org/10.1109/IJCNN.2013.6707043
6. Ban, T., Isawa, R., Guo, S., Inoue, D., Nakao, K.: Efficient malware packer identification using support vector machines with spectrum kernel. In: Eighth Asia Joint Conference on Information Security, AsiaJCIS 2013, Seoul, Korea, July 25-26, 2013. pp. 69–76. IEEE (2013), https://doi.org/10.1109/ASIAJCIS.2013.18
7. Bat-Erdene, M., Kim, T., Li, H., Lee, H.: Dynamic classification of packing algorithms for inspecting executables using entropy analysis. In: 8th International Conference on Malicious and Unwanted Software: "The Americas", MALWARE 2013, Fajardo, PR, USA, October 22-24, 2013. pp. 19–26. IEEE Computer Society (2013), https://doi.org/10.1109/MALWARE.2013.6703681
8. Bat-Erdene, M., Kim, T., Park, H., Lee, H.: Packer detection for multi-layer executables using entropy analysis. Entropy **19**(3), 125 (2017), https://doi.org/10.3390/e19030125
9. Bat-Erdene, M., Park, H., Li, H., Lee, H., Choi, M.: Entropy analysis to classify unknown packing algorithms for malware detection. Int. J. Inf. Sec. **16**(3), 227–248 (2017), https://doi.org/10.1007/s10207-016-0330-4
10. Bergenholtz, E., Casalicchio, E., Ilie, D., Moss, A.: Appendices for: Detection of metamorphic malware packers using multilayered lstm networks. https://github.com/erikbergenholtz/appendix_metamorphic_packers/blob/master/appendix.pdf (2020), accessed: 2020-04-14
11. Blazytko, T., Contag, M., Aschermann, C., Holz, T.: Syntia: Syntesizing the semantics of obfuscated code. In: Proceedings of 26 USENIX Security Symposium. Vancouver, BC, Canada (Aug 2017)
12. Brosch, T., Morgenstern, M.: Runtime packers: The hidden problem. Black Hat USA (2006)
13. Burgess, C.J., Kurugollu, F., Sezer, S., McLaughlin, K.: Detecting packed executables using steganalysis. In: 5th European Workshop on Visual Information Processing, EUVIP 2014, Villetaneuse, Paris, France, December 10-12, 2014. pp. 1–5. IEEE (2014), https://doi.org/10.1109/EUVIP.2014.7018361
14. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/csTRcgi.pl?serial (01 1997)
15. Driller, T.M.: Metamorphism in practice or "how i made metaphor and what i've learnt". https://web.archive.org/web/20070602061547/http://vx.netlux.org/lib/vmd01.html (February 2002), accessed: 2019-12-10
16. Gupta, N., Naval, S., Laxmi, V., Gaur, M.S., Rajarajan, M.: P-SPADE: GPU accelerated malware packer detection. In: Miri, A., Hengartner, U., Huang, N., Jøsang, A., García-Alfaro, J. (eds.) 2014 Twelfth Annual International Conference on Privacy, Security and Trust, Toronto, ON, Canada, July 23-24, 2014. pp. 257–263. IEEE Computer Society (2014), https://doi.org/10.1109/PST.2014.6890947

17. holy_father: Morphine v2.7. https://github.com/bowlofstew/rootkit.com/tree/master/hf/Morphine27 (2004), accessed: 2018-10-24
18. Hubballi, N., Dogra, H.: Detecting packed executable file: Supervised or anomaly detection method? In: 11th International Conference on Availability, Reliability and Security, ARES 2016, Salzburg, Austria, August 31 - September 2, 2016. pp. 638–643. IEEE Computer Society (2016), https://doi.org/10.1109/ARES.2016.18
19. Jeong, G., Choo, E., Lee, J., Bat-Erdene, M., Lee, H.: Generic unpacking using entropy analysis. In: 5th International Conference on Malicious and Unwanted Software, MALWARE 2010, Nancy, France, October 19-20, 2010. pp. 98–105. IEEE Computer Society (2010), https://doi.org/10.1109/MALWARE.2010.5665789
20. Kancherla, K., Donahue, J., Mukkamala, S.: Packer identification using byte plot and markov plot. J. Computer Virology and Hacking Techniques **12**(2), 101–111 (2016), https://doi.org/10.1007/s11416-015-0249-8
21. Kroustek, J., Matula, P., Kolár, D., Zavoral, M.: Advanced preprocessing of binary executable files and its usage in retargetable decompilation. International Journal on Advances in Software **7**(1), 112–122 (2014)
22. Lee, W.Y., Saxe, J., Harang, R.: SeqDroid: Obfuscated Android Malware Detection Using Stacked Convolutional and Recurrent Neural Networks, pp. 197–210. Springer International Publishing, Cham (2019), https://doi.org/10.1007/978-3-030-13057-2_9
23. Morgenstern, M., Pilz, H.: Useful and useless statistics about viruses and anti-virus programs. In: Proceedings of the CARO Workshop (2010)
24. Naval, S., Laxmi, V., Gaur, M.S., Vinod, P.: SPADE: signature based packer detection. In: Chandrasekhar, R., Tanenbaum, A.S., Rangan, P.V. (eds.) First International Conference on Security of Internet of Things, SECURIT '12, Kollam, India - August 17 - 19, 2012. pp. 96–101. ACM (2012), https://doi.org/10.1145/2490428.2490442
25. Rolles, R.: Unpacking virtualization obfuscators. In: Proceedings of USENIX WOOT. Montreal, Canada (Aug 2009)
26. Sun, L., Versteeg, S., Boztas, S., Yann, T.: Pattern recognition techniques for the classification of malware packers. In: Steinfeld, R., Hawkes, P. (eds.) Information Security and Privacy - 15th Australasian Conference, ACISP 2010, Sydney, Australia, July 5-7, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6168, pp. 370–390. Springer (2010). https://doi.org/10.1007/978-3-642-14081-5, https://doi.org/10.1007/978-3-642-14081-5_23
27. Szor, P.: The art of computer virus research and defense. Addison-Wesley, Boston, Mass. (2005)
28. Ugarte-Pedrero, X., Balzarotti, D., Santos, I., Bringas, P.G.: Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015. pp. 659–673. IEEE Computer Society (2015), https://doi.org/10.1109/SP.2015.46
29. Xie, P., Liu, X., Yin, J., Wang, Y.: Absent extreme learning machine algorithm with application to packed executable identification. Neural Computing and Applications **27**(1), 93–100 (2016), https://doi.org/10.1007/s00521-014-1558-4
30. Yan, W., Zhang, Z., Ansari, N.: Revealing packed malware. IEEE Security & Privacy **6**(5), 65–69 (2008), https://doi.org/10.1109/MSP.2008.126