McBoost: Boosting Scalability in Malware Collection and Analysis Using Statistical Classification of Executables

Roberto Perdisci[1,3], Andrea Lanzi[2,3], Wenke Lee[3,1]
{perdisci@damballa.com, andrew@idea.sec.dico.unimi.it, wenke@cc.gatech.edu}

[1]Damballa, Inc. Atlanta, GA 30308, USA
[2]Dip. di Informatica e Comunicazione, Universitá degli Studi di Milano, Italy
[3]College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA

Abstract

In this work, we propose Malware Collection Booster (McBoost), a fast statistical malware detection tool that is intended to improve the scalability of existing malware collection and analysis approaches. Given a large collection of binaries that may contain both hitherto unknown malware and benign executables, McBoost reduces the overall time of analysis by classifying and filtering out the least suspicious binaries and passing only the most suspicious ones to a detailed binary analysis process for signature extraction.

The McBoost framework consists of a classifier specialized in detecting whether an executable is packed or not, a universal unpacker based on dynamic binary analysis, and a classifier specialized in distinguishing between malicious or benign code. We developed a proof-of-concept version of McBoost and evaluated it on 5,586 malware and 2,258 benign programs. McBoost has an accuracy of 87.3%, and an Area Under the ROC curve (AUC) equal to 0.977. Our evaluation also shows that McBoost reduces the overall time of analysis to only a fraction (e.g., 13.4%) of the computation time that would otherwise be required to analyze large sets of mixed malicious and benign executables.

1 Introduction

Malicious executables pose a significant threat to the Internet. As a consequence of malware infection, a victim machine may unintentionally expose sensitive information, participate in remotely coordinated large scale attacks, become a spam sender, host phishing websites, etc. Malware analysis techniques are used for understanding the behavior of malicious executables and extracting signatures useful for detection and containment.

The first step in malware analysis is collecting new malware samples. Existing malware collection tools usually rely on honeypots [26], spam traps [14], and other passive techniques. However, these techniques are "slow" because they require waiting until a new malware starts propagating on a large scale before collecting a copy of it. For example, by the time a new malware hits a honeypot it may have already infected a large number of machines. Actively looking for malware in the Internet has been proposed for example in [21], although in this case the search is restricted to a limited number of suspicious URLs that are considered to be responsible for *drive-by* malware downloads. Crawling P2P networks or the Web is one alternative to the more traditional honeypots and spam traps, and may help reduce the time it takes to collect a new piece of malware. This is true in particular for malware that spread mainly via P2P [19, 7]. Another collection strategy may be to use an *executables sniffer*, which may be deployed at the edge of a network to "sniff" the PE (portable executable) executables that the users of the monitored network are downloading [22]. Considering the disadvantages of traditional malware collection approaches described above (e.g. honeypots and spamtraps), these alternative and more active collection strategies are attractive because they help reduce the time it takes to discover new malware. However, crawling P2P networks or the Web, and "sniffing" executables may result in a very large collection of binaries (e.g., several thousands) that contains a small number of hitherto unknown malware and a high number of benign executables. This would overwhelm most existing binary analysis approaches, for example [8, 24], because they typically need to run each executable for several minutes to understand its behavior. Therefore, we need a way to quickly and accurately classify the executables into *malware* or *benign*, thus allowing us to filter out the least suspicious binaries and focus the analysis on only the most suspicious ones.

Statistical classification of executables provides a way

to quickly classify executables into *malware* or *benign*, and has been explored in a number of works, for example [17, 9, 15, 25]. In particular, n-gram analysis has been shown to be quite successful in detecting malicious executables. However, to the best our knowledge no previous work on statistical malware detection using n-gram analysis takes into account the fact that most malware (92%, according to [4]) use executable packing techniques [18, 20] in order to hide their malicious code. Furthermore, previous works do not consider the fact that some benign executables also use packing techniques in order to protect themselves against violations of commercial licenses. We will show in Section 3 that not taking executable packing into account during the training of classifiers based on n-gram analysis such as the one proposed in [9], for example, may cause their classification accuracy to degrade. In particular, these classifiers may become biased in distinguishing between *packed* and *non-packed* executables, instead of correctly distinguish between *malware* and *benign* executables. As a consequence, packed benign executables would likely cause false positives, whereas non-packed malware may remain undetected.

In this paper, we propose Malware Collection Booster (McBoost), a new fast and accurate statistical malware detection tool that takes executable packing into account, and is intended to improve the scalability of existing malware collection and analysis approaches. Given a large collection of binaries that may contain both hitherto unknown malware and benign executables, McBoost reduces the overall time of analysis by classifying and filtering out the least suspicious binaries and passing the most suspicious ones to a detailed binary analysis process for signature extraction. Figure 1 presents an overview of McBoost. Our system consists of three modules: **A**) A classifier specialized in detecting whether an executable is packed or not; **B**) a universal unpacker based on dynamic binary analysis; and **C**) a classifier specialized in distinguishing between malicious or benign code. If an executable e is deemed packed by module **A**, it will be given to an external unpacker (module **B**) for hidden code extraction, and then the hidden code will be passed to module **C**. In case the unpacker is not able to extract any hidden code (perhaps because e implements strong anti-emulation techniques), e will be added to a list of (likely) "heavily" packed executables which need to be manually inspected (and thus will not be further analyzed by McBoost). The executables are stored into this list along with additional information on what caused the unpacker to fail (e.g., time-out, application crash, malformed PE header, etc.). On the other hand, if the executable is deemed non-packed by module **A**, the code portion of the executable will be directly given to module **C**. Module **C** will then output the probability that the executable being tested contains malicious code.

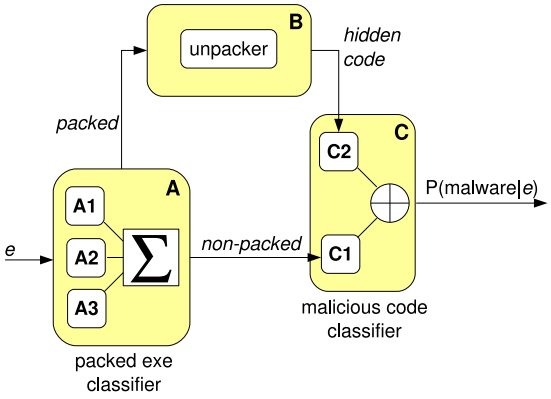

Figure 1: Overview of McBoost Classification System.

It is worth noting that there exists evidence that more than 50% of new malware are simply re-packed versions of already known malware [20]. Therefore, detecting packed malware, and then unpacking them, helps the accuracy of classifier **C** because the extracted hidden code will likely be similar to malicious code seen by **C** during the training phase.

We will show in Section 3 that modules **A** and **C** provide fast and accurate classification. On the other hand, module **B** performs dynamic-analysis-based universal unpacking in a very similar way as [8], and is therefore computationally expensive. However, in the case of malware collection via P2P, Web crawling, or *executable sniffing*, we expect the percentage of *zero-day* malware to be fairly small compared to the benign executables (note that here we assume the known malware have already been filtered out using signature-based AV-software). Given that most benign executables are non-packed, the majority of the collected executables will be quickly and accurately classified, and be passed directly from module **A** to module **C** without the need to attempt a time-consuming unpacking process. Therefore, McBoost provides a fast way to correctly classify and filter out most of the benign executables, thus decreasing the workload of the tools (and humans) that are responsible for performing further detailed binary analysis. This makes the collection and analysis of new malware scalable in the presence of large sets of mixed (unknown) malware and benign executables. Without a system like McBoost this process would take too much time, because all the collected executables would need to undergo a time-consuming analysis, and even by using sophisticated binary analysis tools (e.g., [24]) the average time needed to analyze and classify each single executable is still very high (e.g., several minutes).

Modules **A** and **C** both use n-gram analysis for different classification purposes, as we discuss in Section 2. We would like to emphasize the fact that although n-gram anal-

ysis for malware detection has already been used in a number of previous works [9, 15, 25], we apply n-gram analysis in a novel way. Differently from other works, we recognize that not taking into account the fact that most malware are packed may degrade the classification accuracy of previously proposed malware detection approaches (see Section 3). Therefore, we propose to use n-gram analysis in combination with heuristics approaches in order to accurately distinguish between packed and non-packed executables, first. Then, we study the effectiveness of n-gram analysis for the detection of hidden code extracted from packed executables. The hidden code extracted using universal unpacking may only partially include the original malicious code, as we discuss in Section 2.2. However, we show in Section 3 that even in this case n-gram analysis may still allow us to detect malicious code with relatively high accuracy. To the best of our knowledge, this is also a new result.

We developed a proof-of-concept version of McBoost and evaluated it on 5,586 distinct known malware from the Malfease dataset (`http://malfease.oarci.net`) and 2,258 benign extracted from an installation of Windows XP Home with the addition of common user applications (e.g., WinZIP, WinAmp, AcroRead, etc.). We obtained a classification accuracy of 87.3% (using a detection threshold equal to 0.5) and an Area Under the ROC curve (AUC) equal to 0.977. The AUC can be interpreted as the probability of scoring malware executables higher than benign executables [5], and shows that our classification approach is very promising. Our evaluation also shows that the total time it takes for modules **A** and **C** to classify an PE executable is as low as 1.06 seconds, on average. This means that McBoost is able to quickly filter out the least suspicious executables, thus reducing the number of binaries that need to undergo detailed, time-consuming, binary analysis for studying possible malicious behavior and extracting a detection signature. Therefore, McBoost reduces the overall time of analysis to only a fraction (e.g., 13.4%) of the computation time that would be otherwise required to analyze large sets of mixed malicious and benign executables.

The remainder of the paper is organized as follows. We present the details of our McBoost classification system in Section 2. In Section 3 we report the experimental results. We discuss the most relevant related work in Section 4, and then we briefly conclude in Section 5.

2 McBoost's Internals

McBoost consists of three modules, namely module **A**, **B**, and **C**, as shown in Fig. 1. In this section we describe the internals of each single module.

2.1 Detecting Packed Executables

Module **A** performs detection of packed executables using a Multiple Classifier System (MCS) [10] that combines three classifiers (see Fig. 1). The first classifier is based on a number of heuristics on the structure of the PE file (**A1**) as proposed in [13]. The second classifier is based on n-gram analysis of the code portion of the executable (**A2**), while the third classifier is based on n-gram analysis of the entire binary (**A3**). Although each single module has already high detection accuracy (see Section 3), we combine module **A1** to **A2** and **A3** in order to make the classification of packed executables even more accurate and robust against evasion.

The design and implementation of module **A2** and **A3**, and the MCS that combines them with **A1** to improve on the accuracy and robustness of single classifiers is one of the contributions of this work.

2.1.1 Heuristics-based Classifier

Module **A1** performs classification of packed vs. non-packed executables using a number of heuristics extracted from the structure of PE files. We use the approach proposed in [13]. We extract nine features from the PE file, namely: 1) Number of standard sections; 2) Number of non-standard sections; 3) Number of Executable sections; 4) Number of Readable/Writable/Executable sections; 5) Number of entries in the IAT; 6) Entropy of the PE header; 7) Entropy of the code (i.e., executable) sections; 8) Entropy of the data sections; 9) Entropy of the entire PE file.

A Multi-Layer Perceptron (MLP) classifier that uses these features is trained to distinguish between packed and non-packed executables, as we describe in Section 3. We chose to use MLP because this classification algorithm was shown to achieve better accuracy and generalization ability than other classification algorithms and signature-based detection approaches [13].

2.1.2 n-gram-based Classifiers

Modules **A2** and **A3** perform n-gram analysis of the code section and of the entire PE file, respectively. Each PE executable can be seen as a binary string s. For module **A2**, s represents only the code section, whereas for **A3** it represents the entire file. Given a training dataset of packed and non-packed executables, we create a dataset of binary strings $\mathcal{S}^{(\mathcal{P})} = \{s_i^{(\mathcal{P})}\}_{i=1..k}$ derived from the packed executables, and a dataset $\mathcal{S}^{(\mathcal{N})} = \{s_i^{(\mathcal{N})}\}_{i=1..l}$ derived from the non-packed executables, and we call $\mathcal{S} = \mathcal{S}^{(\mathcal{P})} \cup \mathcal{S}^{(\mathcal{N})}$.

An n-gram is defined as an n-bytes-long substring of a binary string s. We extract all the possible n-grams from each string $s \in \mathcal{S}$, and then we select the M most informative n-grams, i.e., the most discriminative (or "powerful")

features that allow us to separate instances of the positive and negative class, according to an information gain metric [23]. Specifically, the information gain of an n-gram g is

$$\begin{aligned} IG(g) &= \sum_c P(c)\frac{1}{\log P(c)} \\ &+P(g)\sum_c P(c|g)\log P(c|g) \\ &+P(\bar{g})\sum_c P(c|\bar{g})\log P(c|\bar{g}) \end{aligned}$$

where $c \in \{\mathcal{P}, \mathcal{N}\}$ represents either the class of packed executables $\mathcal{P}$ or the class of non-packed executables $\mathcal{N}$, $P(c|g)$ is the probability of a string s (i.e., an executable) being of class c given that g is present in s, and $P(c|\bar{g})$ is the probability of a string s being of class c given that g is not present in s.

Once the M most informative n-grams have been selected, each executable e can be translated into a binary vector $f(e) = x \in \{0,1\}^M$, where the i-th element x_i tells us whether the string s that represents the executable (either its code or the entire file) contains the i-th most informative n-gram ($x_i = 1$) or not ($x_i = 0$). Using this approach, $\mathcal{S}$ can be translated into a labeled dataset of pattern vectors that can in turn be used to train a statistical classifier.

We use a Bagged-Decision-Tree (BDT) classifier [3] for both **A2** and **A3**. Bagged-Decision-Trees usually perform very well and have the characteristic of producing calibrated posterior class probabilities [11], which may be useful when combining multiple classifiers (see Section 2.1.3). The details regarding the algorithm and parameters we used for training the classifiers are reported in Section 3.

The utility of n-gram analysis for the code section (module **A2**) is intuitive. The average length of the instructions of Windows executables for x86 processors is between 2 and 3 bytes (we measured it on the code section of thousands of benign executables). Choosing $n \geq 2$, the presence or absence of an informative n-gram is related to the presence or absence of a certain instruction in the code, or a sequence of instructions, given that an n-gram may capture the end of an instruction and the beginning of the next, for example. Therefore, each element of a pattern vector $f(e) = x \in \{0,1\}^M$ given to the classifier is related to the presence or absence of an instruction or sequence of instructions in the code of an executable e. Since the code section of a packed executable usually contains the unpacking/decryption routine and (possibly) the hidden (encrypted) code, its distribution of n-grams will likely be different from the distribution of n-grams of a non-packed executable. Module **A2** is designed to detect this difference in the distribution of n-grams.

The utility of n-gram analysis for the entire file (module **A3**) is also intuitive. If the hidden (encrypted) code is stored in a data section, hidden in unused fields of the main and section headers, or somewhere else in the file, module **A2** may not detect the fact that the executable is packed. The n-gram analysis on the entire file adds a piece of evidence for making a final decision, and is based on the fact that the presence of encrypted code in the file will cause a perturbation of the overall "normal" distribution of n-grams (i.e., the distribution of n-grams of non-packed executables). Therefore, module **A3** tries to capture this perturbation.

2.1.3 Combining Multiple Classifiers

For each input executable e, the output of each of the modules **A1**, **A2** and **A3** is a posterior class probability $P_{A_i}(\mathcal{P}|e)$, i.e., the probability that e is a packed executable as estimated by module **Ai**, with **i=1**, **2**, **3**. Module **A** is a Multiple Classifier System (MCS) [10] that combines the output of **A1**, **A2**, and **A3**, and produces an overall posterior class probability $P_A(\mathcal{P}|e)$ of e being packed.

Multiple Classifier Systems usually perform better than the single classifier in the ensemble. This is particularly true when the classifiers are *diverse*, in the sense that they make different (ideally independent) mistakes on different input vectors (i.e., different representations of PE executables, in our case) [10]. Diversity may be induced in a number of ways [6]. In our application we introduced diversity among **A1**, **A2**, and **A3** by training them on diverse representations of the same dataset of packed and non-packed executables. Also, we used different classification algorithms (Multi-Layer Perceptron for **A1** and Bagged-Decision Trees for **A2** and **A3**). Another important characteristic of MCS is that the output of the single classifiers should be comparable [6]. We chose Multi-Layer Perceptron and Bagged-Decision-Tree as base classifiers because they both output well-calibrated posterior class probabilities [11]. Also, MCS are more robust and have been shown to be useful in making evasion by mimicry attacks harder [12]

We use a simple but effective combination rule to combine the output of **A1**, **A2**, and **A3**, namely the average of probabilities [10] and we set a decision threshold θ so that if $P_A(\mathcal{P}|e) = \frac{1}{3}\sum_{i=1..3} P_{A_i}(\mathcal{P}|e) > \theta$ the executable e is classified as *packed* (and sent to the unpacker), otherwise e is classified as *non-packed* (and sent directly to module **C**). θ can be tuned in order to find the desired trade-off between the false positive and false negative rates for module **A**.

2.2 Extracting Hidden Code

We implemented our unpacker (module **B**) in a way very similar to Renovo [8], using the QEMU emulator [1]. Every time an instruction i is executed on behalf of a process P, the unpacker U will intercept it and check whether i was previously dynamically generated by P itself. If this is the

case, i will be marked as *hidden* code. i will then be considered as the first instruction of a *layer* of unpacking [8]. Whenever U detects that P is trying to execute a new dynamically generated instruction i', this will mark the end of a layer of unpacking and the start of a new one. U will then dump the binary code of the first layer to disk and will keep tracing the execution of the next layer [8].

We adopt two different strategies to dump the *hidden* binary code in each layer of unpacking:

- ***bpage***. This strategy dumps all the memory pages where the instructions belonging to a hidden layer reside. For example, assume instruction i_1 belongs to an unpacking layer l and is located in page p_1, and another instruction i_2 also belongs to the same layer l, but is located in page p_2. In this case both the p_1 and p_2 (the entire pages) will be dumped by U and marked as related to the layer l.

- ***bbexec***. This strategy dumps only the instructions of a layer of unpacking that were actually executed by P. In order to do this, the monitor U keeps trace of the instructions in the QEMU translation blocks [1] (or basic blocks) that were actually executed by the emulator.

The *bbexec* dumps may be useful in those cases when a binary is packed using executable packing tools that perform encryption/decryption operations at the single instruction level. On the other hand, the *bpage* dumps may be useful to extract the hidden code of binaries that were packed by executable packing tools that perform encryption/decryption at the memory page level. It is worth noting that both techniques may provide only a partial reconstruction of the original executable code embedded in the packed binary.

We set two time-out parameters, namely a *per layer* time-out T_l and a *global* time-out T_g. If any time-out is reached the process P will be terminated. Apart from reaching a time-out, there are other reasons why U may terminate the process P. For example, our unpacker is able to intercept calls to the `NtRaiseHardError` native API and therefore report an application crash. Also, the unpacker is able to detect "normal" process exits and system errors.

2.3 Detecting Malicious Code

Similarly to module **A2**, the classifiers in module **C** are based on the n-gram analysis of the code portion of executables. The difference is in the fact that module **A2** is specialized in detecting packed vs. non-packed code, whereas modules **C1** and **C2** are responsible for distinguishing between malware vs. benign non-packed or hidden (extracted by the unpacker) code, respectively. The design and implementation of specialized classifiers based on n-gram analysis of non-packed code (**C1**) and hidden code extracted from packed executables (**C2**) is one of the contributions of this work.

In order to train module **C1** we collect a dataset of non-packed malware and non-packed benign executables (the details of how we construct the dataset are reported in Section 3.1.1), and extract their code sections. We then select the M most informative n-grams in the code of executables from the two classes, as explained in Section 2.1.2, and we use these n-grams to translate each executable into a pattern vector representation that can be used to train a statistical classifier. As for modules **A1** and **A2**, we use Bagged-Decision-Trees (BDT) as classifier. During test, for each analyzed executable e the output of module **C1** (i.e., of the BDT classifier) is an estimate of the probability $P(malware|e)$ that e's (non-packed) code is malicious.

We use the same approach to train module **C2**. The only difference is that the training dataset is made of a collection of hidden-code extracted (using our unpacker) from packed malware and packed benign executables. Like for **C1**, the output of **C2** is an estimate of the probability $P(malware|e)$ that e's hidden code is malicious. The use of either module **C1** or module **C2** for each executable e under test depends on the results of modules **A** and **B** (i.e., of the packer detector and the unpacker), as explained above (at the beginning of Section 2) and shown in Fig. 1.

The intuition behind the use of n-gram analysis is that the presence or absence of an informative n-gram (see Section 2.1.2) is related to the presence or absence of a certain instruction or sequence of instructions in the code of an executable e. We speculate that the code section of malicious executables contain certain instructions, or sequences of instructions more than others. These instructions are used to carry out malicious activities and may not be present with the same frequency or sequence in benign code. This intuition is (partially) in accordance with the results reported in [2]. Modules **C1** and **C2** capture this difference in the distribution of sequences of instructions between malicious and benign code using n-gram analysis.

3 Experiments

In this section we discuss in detail how we performed the evaluation of McBoost and its components, and we present the obtained experimental results. We performed our experiments on a machine with a 2GHz dual-core AMD Opteron processor and 8 GByte of memory.

3.1 Experimental Setup

3.1.1 Preparation of the Datasets

To evaluate the effectiveness of McBoost, we collected several thousands of Windows benign and malicious PE exe-

cutables. Overall we collected 5,586 distinct known malware binaries and 2,258 benign, which we divided in the following labeled datasets:

Malware-Dataset (MDset). We collected a set of 5,586 malware executables from the Malfease dataset (`http://malfease.oarci.org`) in July 2007. We used three Anti-virus (AV) software products, namely clamAV (`www.clamav.net`), F-Prot (`www.f-prot.com`), and AVG (`free.grisoft.com`), to verify that the binaries collected from the Malfease dataset were actually all know malware.

Packed-Malware-Dataset (PMDset). We used PEiD (`http://www.peid.info`) and the packer-detection capabilities of F-Prot to select packed malware binaries from Malware-Dataset, and we obtained 2,078 packed binaries. PEiD detected 2,039 binaries packed using around 70 distinct packers, whereas F-Prot detected 328 binaries packed using 20 distinct packers. The two sets slightly overlap. For around one third of the malware detected as packed by F-Prot, the use of multiple layers of packing was reported (to the best of our knowledge, PEiD is not capable of detecting multi-layer packing).

Non-Packed-Malware-Dataset (NPMDset). We filtered out the binaries in Packed-Malware-Dataset from the Malware-Dataset, thus keeping 3,508 binaries. On this set we ran Polyunpack [16] and our dynamic unpacker and found 174 executables for which neither Polyunpack nor our unpacker were able to extract any hidden code, and that did not cause any error (e.g., application crash). On these 174 executables we also ran Renovo[1] [8] in order to further filter any possibly packed executable missed (i.e., no hidden code was detected) by Polyunpack and our unpacker[2]. We filtered-out the binaries for which Renovo was able to extract any hidden code and we obtained 146 likely non-packed malware (although this dataset may still contain few packed executables, we believe the use of multiple state-of-the-art techniques allowed us to reduce possible noise to a minimum).

Other-Malware-Dataset (OMDset). This dataset consists of the 3,362 malware in MDset that do not belong to neither PMDset nor NPMDset. Although many of this executables are likely packed (because at least one of the three universal unpackers we used was able to extract some kind of hidden code from them), they were not detected by the signature-based packer detectors, i.e., PEiD and F-Prot. Therefore, we chose not to include them in the PMDset because we did not want to risk to "poison" the PMDset with the possible false positives caused by dynamic unpackers [3] (i.e., either Polyunpack, our unpacker, or Renovo).

Benign-Dataset (BDset). We collected 2,258 benign executables extracted from an installation of Windows XP Home with the addition of common user applications (e.g., WinZIP, WinAmp, AcroRead, etc.). We double checked these binaries using clamAV, F-Prot, and AVG to verify that the collected binaries were actually benign applications. We also submitted the binaries for which we had any doubts to VirusTotal (`www.virustotal.com`) to check them against a diverse set of AV-software.

Packed-Benign-Dataset (PBDset). We ran PEiD on Benign-Dataset and we found 27 packed binaries (i.e., around 1.2% of the benign), most of which were packed with UPX. Also, we selected 45 benign executables from the start menu of a clean Windows XP installation and we packed each one of them with 17 different popular packers (including UPX (`upx.sourceforge.net`), ASpack (`www.aspack.com`), Themida (`www.oreans.com/themida.php`), Obsidium (`www.obsidium.de`), etc.) That is, for each of these executables, we created 17 packed executables. For each packer we enabled all of the available anti-debugging and anti-reverse engineering techniques. We verified that some packers failed to correctly pack several binaries, causing them to fail when executed. Therefore, we pruned the dataset keeping 195 binaries that still worked correctly after packing. Overall, we obtained 222 (195 plus 27) packed benign executables.

Non-Packed-Benign-Dataset (NPBDset). This dataset was obtained by removing the 27 packed benign we found in Benign-Dataset, thus keeping 2,231 non-packed benign.

3.1.2 Parameter Setting

In this section we discuss how we set the parameters of each module of McBoost. For constructing the classifiers we use WEKA (`http://www.cs.waikato.ac.nz/ml/weka`), a collection of open-source data mining software written in Java. For module **A1** (see Section 2) we use a Multi-Layer Perceptron (MLP) classifier. Our MLP has three layers, namely an input layer, a hidden layer and an output layer. The input layer has 9 nodes (equal to the number of features extracted by **A1**), whereas the output layer

[1] We were able to do this thanks to the kind collaboration of the authors of Renovo.

[2] Although our universal unpacker follows the design described in [8], some implementation details may be different, and therefore the result of unpacking may differ from the result obtained with Renovo, in some cases.

[3] We found that some executables dynamically generate few executable instruction in memory and then jump on them, although they did not appear to be packed with executable packing tools, according to our manual analysis.

Classifier	Accuracy	FP	DR	AUC
A1 (heuristics)	0.973	0.012	0.958	**0.995**
A2 (n-gram on code)	0.976	0.034	0.987	**0.981**
A3 (n-gram on file)	0.993	0.004	0.990	**0.993**
A (multiple classifiers)	0.994	0.008	0.996	**0.997**

Table 1: Validation of module **A** for *packed* vs. *non-packed* classification using a detection threshold = 0.5 (FP = false positive rate, DR = detection rate).

Classifier	Accuracy	FP	DR	AUC
C1 (non-packed code)	0.823	0.026	0.772	**0.959**
C2 *bpage* (hidden code)	0.938	0.0	0.937	**0.988**
C2 *bbexec* (hidden code)	0.745	0.112	0.738	**0.901**

Table 2: Validation of module **C1** and **C2** for *malware* vs. *benign* code classification using a detection threshold = 0.5 (FP = false positive rate, DR = detection rate).

has two nodes (one for each class label). We set the number of perceptrons in the hidden layer to 5. Modules **A2**, **A3**, **C1**, and **C2** (see Section 2) are all built using Bagged-J48 (J48 is an implementation of the well-known C4.5 decision-tree classifier). For each module, the Bagged-J48 is constructed by combining 10 base decision-tree (J48) classifiers. We used n-grams with $n = 3$ for **A2**, **A3**, and **C1**, whereas we used $n = 2$ for **C2** because it produced much better results then using $n = 3$. The *per layer* and *global* time-out for our unpacker (module **B**) were set to $T_l = 4$ minutes and $T_g = 20$ minutes, respectively (see Section 2.2).

3.2 Validation of Single Modules

Detection of Packed Executables (Module A). In order to test module **A**, and its sub-modules **A1**, **A2**, and **A3**, we constructed a labeled dataset of packed and non-packed executables. By merging the dataset of packed malware PMDset and the dataset of packed benign PBDset we obtained the dataset of packed executables. The dataset of non-packed executables was constructed by merging the dataset of non-packed benign NPBDset and the dataset of non-packed malware NPMDset. Overall, we obtained a dataset which consisted of 2,300 packed executables and 2,377 non-packed executables. We then randomly split this dataset into two portions, a portion made of 80% of the data used for training the classifiers, and a portion made of the remaining 20% of the data for testing. The classification results are reported in Table 1. The accuracy, false positives and detection rate were computed setting the detection threshold $\theta = 0.5$ (in the following we will always assume $\theta = 0.5$ for module **A**, unless otherwise specified). The average time needed for the classification of an executable was 1.025 seconds. As we can see from Table 1, the combination of classifiers improves on the already very good performance of the single classifiers, in particular in terms of AUC. Also, combining diverse classifiers contributes in making evasion attempts intuitively harder.

Extracting Hidden Code (Module B). In order to validate the performance of our implementation of the dynamic unpacker (module B), we tested it with the executables in the PMDset (2,078 packed malware) and PBDset (222 packed benign). Our unpacker was able to correctly extract hidden code from 169 packed benign (76.1%) and from 1,943 packed malware (93.5%). Among the 188 packed executables that module B was not able to unpack, 25 (1 benign and 24 malware) caused an "application crash" error, whereas 29 (all malware) caused a "non-win32 application" error (likely because of a corrupted PE header).

We also measured the time needed for the unpacker to analyze an executable. We found that the average time for analyzing a malware executable was 4.7 minutes, whereas the average time for the analysis of a benign executable was 5.6 minutes.

Detection of Malicious Code (Modules C1 and C2). In order to evaluate module **C1** we constructed a labeled dataset of code sections extracted from non-packed benign (NPBDset) and non-packed malware (NPMDset). In total, we had 2,377 non-packed executables. The PE analysis tool we used to extract the content of the code section failed in certain cases, either because the PE header was found to be corrupted, or because the PE file did not contain any section marked as executable. After filtering out these cases we obtained a labeled dataset of 2,357 code sections, 2,229 extracted from non-packed benign and 128 extracted from non-packed malware. We randomly split this dataset in two parts while maintaining the proportions between the two classes (malware and benign). We used 80% of the dataset for training the classifier, and 20% for testing. The results are reported in Table 2 for a detection threshold equal to 0.5.

We evaluated the classification accuracy of **C2** in a way similar to **C1**. We ran our dynamic unpacker on the entire dataset of malware (MDset) and on the dataset of packed benign (PBDset). We first considered *bpage* dumps of the last layer of unpacking (see Section 2.2). We obtained 3,856 *bpage* dumps from malware samples and 169 *bpage* dumps from the packed benign. Given this dataset of labeled (*malware* or *benign*) *bpage* dumps, we randomly split (maintaining the proportion between the two classes) in two parts. We used 80% of the dataset for training purposes and the remaining 20% for testing. The results are reported in Table 2, second row. The third row in Table 2 reports the results of a similar experiment using the *bbexec* dumps of the last layer of unpacking (see Section 2.2), instead of the *bpage* dumps. It is easy to see that using *bpage* dumps provides better re-

sults, therefore in the following we only consider the results obtained using *bpage* dumps.

The average time needed for the classification of an executable by module **C** was 0.032 seconds.

3.3 Validation of McBoost

In order to validate the ability of our McBoost system to correctly detect and rank previously unknown malicious executables, we randomly chose 80% of the patterns in the labeled datasets PMDset, NPMDset, PBDset, and NPBDset for training the single classifiers in our system (i.e., **A1**, **A2**, **A3**, **C1**, and **C2**). We then used the remaining 20% of these datasets plus the entire OMDset for testing. Overall, the test dataset contained 3,830 malware and 503 benign executables. Module **A** classified 2,471 of these executables as packed, and they were sent to the unpacker for extracting the hidden code. The remaining 1,862 executables were classified as non-packed. The unpacker was able to extract the hidden code from 1,441 out of 2,471 packed executables (58.3%).

Therefore, 1,862 executables were sent to module **C1**, 1,441 were sent to **C2** (i.e., 3,303 executables were sent to module **C**, in total), and 1,030 where stored in the list of (likely) "heavily" packed executables that need manual inspection. Among these 1,030 executables, 8 were packed benign and the remaining 1,022 were malware. We found that 613 out of these 1,022 malware caused an application crash during the unpacking process, whereas 60 of them generated a "non-win32 application" error message (likely because of a corrupted PE header).

The results of the classification of the 3,303 executables sent to modules **C** (either to **C1** or **C2**) are reported in Table 3. Table 3 reports the values of detection threshold, false positives, detection rate and accuracy for significant points on the ROC curve. The Area Under the ROC curve (AUC) is equal to 0.977. It is also worth noting that if we set the detection threshold to 0.902, McBoost is still able to correctly detect 61.7% of the malware with no false positives, i.e., no benign executable will be considered for further (possibly manual) analysis. On the other hand, if we are willing to accept some false positives, i.e., if we can afford to perform a detailed, manual analysis of more executables, but we do not want many false negatives because we are not willing to miss many unknown malware, we can set the threshold to 0.007 and obtain 99.3% detection rate with 27% of false positives.

The average time needed for classifying an executable found to be non-packed by module **A** was around 1.06 seconds in average (1.025 sec. for module **A** and 0.032 sec. for module **C**), whereas the average time needed to classify an executable sent to the unpacker was around 4.7 minutes for malware and 5.6 minutes for benign executables.

Threshold	False Positive Rate	Detection Rate	Accuracy
0.902	0	0.617	0.673
0.686	0.010	0.836	0.859
0.500	*0.025*	*0.856*	*0.873*
0.284	0.050	0.881	0.891
0.126	0.100	0.916	0.913
0.029	0.200	0.980	0.953
0.007	0.270	0.993	0.954

Table 3: Significant values of the trade-off between false positives, detection rate, and accuracy for different values of the detection threshold of McBoost. The AUC is 0.977.

Classifier	% Unpacked	Accuracy	FP	DR	AUC
McBoost	57.9%	0.870	0.0	0.727	**0.930**
KM	-	0.429	0.875	0.833	**0.472**

Table 4: Comparison between McBoost and the approach presented in [9] on a difficult dataset (FP = false positive rate, DR = detection rate).

Comparison to Previous Work. Table 4 shows the comparison between McBoost and the approach proposed by Kolter et al in [9], which we called KM. We implemented the classifier proposed in [9] according to the description given in the paper, and we set the same values for the parameters such as the value of n, the number of features to be selected etc., as suggested in the paper. We trained KM similarly to McBoost, using 80% of the PMDset and NPMDset as malware dataset, and 80% of BDset for the benign dataset. Afterwards we tested both McBoost and KM on the same test set consisting of 20% of PBDset (packed benigns) and NPMDset (non-packed malware). The results reported in Table 4 confirm the fact that KM is biased towards detecting packed executables as malware and non-packed executables as benign, regardless of the nature of the hidden or non-packed code. On the other hand, McBoost still has an accuracy of 87% and an AUC of 0.93 even in the case of such a "difficult" dataset.

3.4 Discussion of the Results

We would like to emphasize that the most important result is the value of the AUC of McBoost, which is equal to 0.977 in our experiments, because the AUC can be interpreted as the probability of scoring a malware executable higher than a benign executable [5], in terms of $P(malware|e)$. Since our system is intended for prioritizing (according to the ranking given by McBoost's output, $P(malware|e)$) the analysis of the most suspicious binaries, a value of the AUC close to 1 (which is the maximum possible value) is intuitively more important than the accuracy, whose value is dependent from the *a priori* class probabilities and from the detection threshold.

The utility of McBoost is intuitive. Suppose we need to analyze a large set of executables downloaded by a P2P or web *executable crawler*, or by and *executable sniffer*

in order to collect samples of new (*zero-day*) malware. In this case, once we filter out the known malware using signature-based AV-software, we expect the dataset will contain mostly benign executables and a small fraction of unknown malware. Also, we expect most of the benign to be non-packed (in Section 3.1.1 we found that only 1.2% of the executable of a typical Windows XP home user installation were packed). Given that non-packed executables can be classified in around 1.06 seconds in average, McBoost allows us to quickly filter out most of the benign executables because they will pass from module **A** to module **C** directly, and will receive a low $P(malware|e)$ (i.e., a low rank). On the other hand, existing approaches for analyzing executables downloaded/collected from the Internet most likely have to run each unknown executable (e.g., after running AV tools, and checking a whitelist) through an unpacker and/or a program analyzer. As suggested by our results above, it takes at least several minutes to analyze each executable just to determine whether there is hidden code and if so extract the code. If we assume that the majority of the executables are non-packed benigns, this process is very wasteful and inefficient. Therefore, McBoost can achieve huge time savings when processing a large collection of executables from the Internet. Without McBoost we would need to analyze all of the binaries (after black-list/white-list filtering) using expensive analysis techniques, thus increasing the overall cost of the analysis to the point of infeasibility.

The total processing time for the validation of McBoost using the 4,330 executables (3,830 malware plus 503 benigns, as described above), was about 195 hours (slightly more than 8 days). On the other hand, processing all the 4,330 test samples directly using our dynamic unpacker (i.e., assuming we do not have our module **A** classifier) would take about 347 hours (slightly less than 14 and $\frac{1}{2}$ days). Therefore, McBoost required only 56.2% of the time compared to running all the executables through module **B** or similar binary analysis tools. It is worth noting, though, that our test dataset contained more malware samples than benigns (88.4% of the executables were malware). As mentioned before, by crawling P2P networks (or the Internet in general) looking for executables may very likely produce (after white- and black-listing) a dataset containing a small percentage of new malware and a large percentage of non-packed benigns. In this case we expect the time saving due to McBoost to be much higher. For example, if we collected a dataset that contains about 85% of non-packed benign executables, 2% of packed benign, and 13% of unknown malware (which for the sake of this example we assume all packed), we expect McBoost will require only around 13.4% of the time, compared to using only tools based on dynamic binary analysis similar to module **B**.

4 Related Work

In [16], Royal et al. proposed Polyunpack, an universal unpacker based on a combination of static and dynamic binary analysis. Given a packed PE executable, a static view of the code is constructed first. If the executable tries to execute any code that is not present in the static view, Polyunpack will detect this as an attempt to running hidden code and will try to reconstruct the original executable.

Renovo, a different and somewhat more effective tool for universal unpacking using dynamic binary analysis, was presented by Kang et al. in [8]. Renovo is able to distinguish among different layers of unpacking and dump the memory pages that contain the hidden code for each layer.

In [17] Shultz et al. present data mining techniques for detecting new malicious executables. They extract several features from the executables, for example the list of DLLs used by the binary, the list of DLL function calls, and the number of different system calls used within each DLL. Also they analyze byte sequences extracted from the *hexdump* of an executable (i.e., its hexadecimal representation) [17]. Our work is different from [17] because we adopt a different approach. We first distinguish between *packed* and *non-packed* executables, and then (if needed) we extract and classify the hidden or non-packed code into *malware* or *benign*. Also, we measure a different set of features extracted from PE executables, compared to [17].

N-gram analysis for malware detection has been studied in a number of works [9, 15, 25]. To the best of our knowledge, among these the work closest to ours is [9]. Kolter et al. [9] use n-gram analysis on entire PE files to distinguish between malware and benign executables. However, they do not distinguish between packed and non-packed executables. They collect 1,651 malware samples and 1,971 benign samples [9]. They take their dataset of malware as it is without considering whether the executables they collected are packed or not, and show that their best classifier (Boosted J48) achieves an AUC$\simeq 0.996$. Because most of today's malware are packed [4, 20], not taking this into account during the training and test of the classifier may produce over-optimistic results. In the presence of a training dataset containing mainly packed malware and non-packed benign, the approach of Kolter et al. may actually be biased in distinguishing between packed and non-packed executables, instead of malware vs. benign executables, as we show in Section 3. Although we use n-gram analysis in our McBoost, our approach is significantly different from [9]. We first recognize that most of the malware are packed, and that only distinguishing between packed and non-packed executables may not be enough to actually detect malicious executables. Therefore, we first classify executables into packed and non-packed, and for the packed executables we try to extract the hidden code using an universal unpacker

similar in principle to Renovo [8]. We then classify the extracted hidden code (or the non-packed code, if an executable is found to be non-packed) into either *malicious* or *benign*.

5 Conclusion

We presented McBoost, a fast statistical malware detection tool intended to improve the scalability of existing malware collection and analysis techniques. We discussed how McBoost can be used in case when a large collection of binaries that contains both unknown (zero-day) malware and benign executables needs to be analyzed in order to discover new malware samples for which a detection signature can be written. McBoost allows us to quickly and accurately filter out most of the benign and prioritize further detailed analysis of the remaining suspicious binaries. This allows us to significantly reduce the workload of tools (and humans) that perform detailed binary analysis.

We evaluated the accuracy and performance of the individual modules in McBoost as well as the system as a whole. The results showed that McBoost has an overall classification accuracy of 87.3% and an AUC equal to 0.977. In addition, the running time of McBoost on our test data shows that the overal computation time for analyzing large sets of executables can be reduced to only a fraction (e.g., 13.4%) of the time needed if we only used dynamic-analysis-based tools.

References

[1] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, 2005.

[2] D. Bilar. Opcode as predictors for malware. *International Journal of Electronic Security and Digital Forensics*, 1(2), 2007.

[3] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[4] T. Brosch and M. Morgenstern. Runtime packers: The hidden problem? Presented at Black Hat USA 2006.

[5] C. Cortes and M. Mohri. Confidence intervals for the area under the roc curve. In *NIPS 2004: Advances in Neural Information Processing Systems*, 2004.

[6] R. Duin. The combining classifier: to train or not to train? In *International Conference on Pattern Recognition (ICPR)*, 2002.

[7] A. Kalafut, A. Acharya, and M. Gupta. A study of malware in peer-to-peer networks. In *ACM SIGCOMM conference on Internet measurement*, 2006.

[8] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *WORM '07: Proceedings of the 5th ACM Workshop on Recurring Malcode*, 2007.

[9] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7:2721–2744, 2006.

[10] L. I. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. Wiley-Interscience, 2004.

[11] A. Niculescu-Mizil and R. Caruana. Predicting good probabilities with supervised learning. In *International Conference on Machine Learning (ICML)*, 2005.

[12] R. Perdisci, G. Gu, and W. Lee. Using an ensemble of one-class svm classifiers to harden payload-based anomaly detection systems. In *International Conference on Data Mining (ICDM)*, 2006.

[13] R. Perdisci, A. Lanzi, and W. Lee. Classification of packed executables for accurate computer virus detection. *Patter Recognition Letters*, 29(14):1941–1946, 2008.

[14] M. B. Prince, L. Holloway, E. Langheinrich, B. M. Dahl, and A. M. Keller. Understanding how spammers steal your e-mail address: An analysis of the first six months of data from project honey pot. In *2nd Conference on Email and Anti-Spam (CEAS)*, 2005.

[15] D. K. S. Reddy and A. K. Pujari. N-gram analysis for computer virus detection. *Journal in Computer Virology*, 2(3), 2006.

[16] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[17] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *IEEE Symposium on Security and Privacy*, 2001.

[18] A. Shevchenko. The evolution of self-defense technologies in malware, 2007. `http://www.viruslist.com/analysis?pubid=204791949`.

[19] S. Shin, J. Jung, and H. Balakrishnan. Malware prevalence in the kazaa file-sharing network. In *ACM SIGCOMM Conference on Internet measurement*, 2006.

[20] A. Stepan. Improving proactive detection of packed malware, March 2006. `http://www.virusbtn.com/virusbulletin/archive/2006/03/vb200603-packed.dkb`.

[21] Y. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. T. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *NDSS*, 2006.

[22] T. Werner. PE Hunter, 2007. `http://honeytrap.mwcollect.org/pehunter`.

[23] Y. Yang and J. O. Pedersen. A comparative study on feature selection in text categorization. In *International Conference on Machine Learning (ICML)*, 1997.

[24] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS '07: 14th ACM conference on Computer and communications security*, 2007.

[25] B. Zhang, J. Yin, J. Hao, D. Zhang, and S. Wang. Malicious codes detection based on ensemble learning. In *Autonomic and Trusted Computing (ATC)*, 2007.

[26] J. Zhuge, T. Holz, X. Han, C. Song, and W. Zou. Collecting autonomous spreading malware using high-interaction honeypots. In *ICICS 2007*, 2007.