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

Roberto Perdisci[†], Andrea Lanzi[⋆‡], and Wenke Lee[‡†]

[†]Damballa, Inc. Atlanta, GA 30308, USA
[‡]College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA
[⋆]Dipartimento di Informatica e Comunicazione, Universitá degli Studi di Milano, Italy
{ perdisci@damballa.com , andrew@idea.sec.dico.unimi.it , wenke@cc.gatech.edu }

Abstract

Assume a large collection of binaries, which may consist of both hitherto unknown malware and benign executables, needs to be analyzed in order to detect new malicious code and extract a signature. In this case, it would be very useful to quickly identify only those binaries that are potentially packed and should be given to an universal unpacker (which is typically computationally expensive) for extracting the hidden code. Also, it would be useful to know if a non-packed executable or unpacked code (e.g, extracted by the unpacker) is likely malicious or not. This information can be used to prioritize detailed (possibly manual) analysis of the most suspicious executables. In this work, we propose McBoost (Malware Collection Booster), a fast statistical malware detection tool that is intended to improve the scalability of existing malware collection and analysis techniques. The system 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. The results showed that McBoost has a classification accuracy of 83.4% and an Area Under the ROC curve (AUC) equal to 0.973. The total running time of McBoost on these executables shows a significant time saving (e.g., 86.6%) over most existing analysis approaches.

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 help in understanding the behavior of malicious executables and extracting signatures useful for detection and containment. Unfortunately, modern malware implements a number of evasion techniques, for example, code-obfuscation, polymorphism, encryption, and *packing* [26, 28], which make malware analysis hard and time-consuming. In particular, several easy-to-use packing tools that enforce anti-reverse engineering techniques are available on the Internet. As a consequence, the number of packed malicious executables is growing rapidly. A recent report indicates that 92% of the malware is packed [5]. Furthermore, there exists evidence that more than 50% of new malware are simply re-packed versions of already known malware [28].

Universal unpackers have been recently proposed to automate the extraction of (potentially malicious) code hidden in packed executables [23, 11, 16]. Unfortunately, these unpackers add a significant amount of overhead to the binary analysis task, because they need to instrument the binary to be analyzed and follow its execution in a virtualized or emulated environment. Also, since determining if an executable is packed

or not is undecidable [23], the unpacker may need to execute a program until a predefined (possibly large) time-out is reached.

When a large collection of binaries that contains hitherto unknown malware and benign executables needs to be analyzed in order to detect new malicious code and extract a signature, it would be useful to quickly identify only those binaries that are potentially packed and should be given to an (computationally expensive) unpacker for extracting the hidden code. Also, it would be useful to know if a non-packed executable or unpacked code (e.g., extracted by a universal unpacker) is likely malicious or not. This information can be used to prioritize detailed (possibly manual) analysis of the most suspicious executables. In order to solve this problem, we propose McBoost (Malware Collection Booster), a fast statistical malware detection tool that is intended to improve the scalability of existing malware collection and analysis techniques. Figure 1 presents an overview of McBoost. The 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; **C**) a classifier specialized in distinguishing between malicious or benign code. If an executable 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**. On the other hand, if the executable is deemed non-packed, 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.

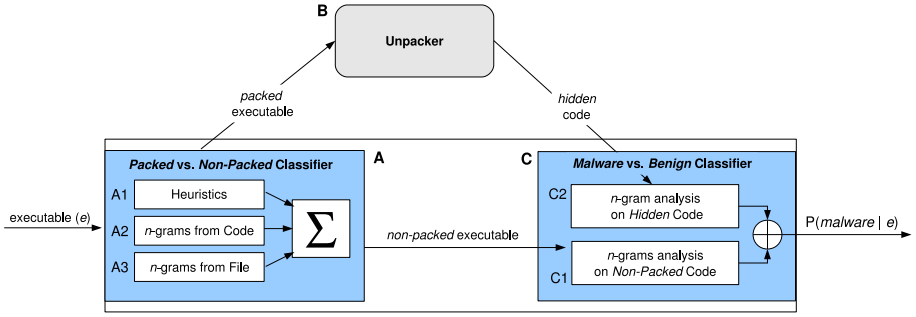

Figure 1: Overview of McBoost Classification System.

An example scenario in which McBoost would be very useful is the fast analysis of binaries downloaded through a P2P (or web) *executable crawler* searching for previously unknown (*zero-day*) malware from which to extract a detection signature. In this case, given the large amount of binaries that may be downloaded by crawling P2P networks [27] (or the Internet as a whole), it would be infeasible to manually (or semi-manually) analyze all of them in a short time, even with the help of sophisticated automatic analysis tools. Therefore, quickly detecting which binaries are the most suspicious, i.e., the most likely to contain malicious code, helps in reducing the amount of time between the discovery of a new malware and the extraction of a detection signature. McBoost can be applied in this case to quickly classify and rank the downloaded executables based on a suspiciousness score, namely the probability $P(malware|e_i)$ of each executables e_i being malware.

Existing malware collection tools usually rely on honeypots [34], spam traps [21], 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 [32], 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 in general) is an alternative to the more traditional honeypots and spam traps, and may help in reducing the time it takes to collect a new piece of malware. This may be true in particular for

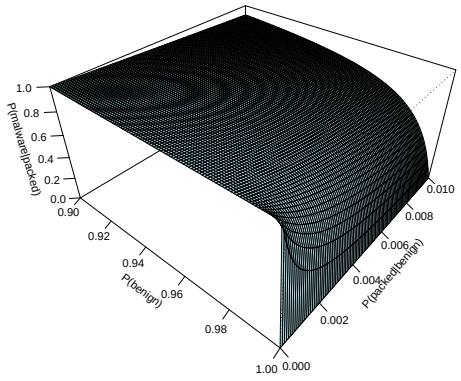

Figure 2: Probability of a packed executable being a malware, assuming $P(packed|malware) = 0.92$, $P(packed|benign) \in [0, 0.15]$, and $P(benign) = 1 - P(malware) \in [0.9, 1]$.

malware that spread mainly via P2P [27, 10]. As mentioned above, this is only feasible if we have a fast way to filter-out the binaries that are already-known malware or are likely non-malicious, so that we can (manually or semi-manually) analyze only the most suspicious (and therefore promising from the point of view of signature extraction) among the remaining ones. Of course, filtering already-known malware can be easily accomplished by using current signature-based approaches. On the other hand, making a decision on the suspiciousness of an executable that is not detected by any AV-software is not so easy. Given the high number of false negatives (or missed detections) of heuristics-based detection of traditional AV-software (up to 80% according to [25]), and the increasing number of new or re-packed malware [28], the fact that an executable is not detected as known-malware by traditional AV-software does not allow us to decide that it is not malicious. We will show that McBoost allows us to make such decision quickly and with high accuracy.

We would like to emphasize that simply detecting whether an executable is packed or not is not sufficient to make a decision on the suspiciousness of the executable itself. Although it has been estimated that over 92% or the malware are packed [5], Figure 2 shows that for plausible values of the probability $P(packed|benign)$ of a benign executable being packed and the *a priori* probability $P(benign)$ of a generic executable being benign, $P(malware|packed)$, the probability of an executable being malicious given that we know it is packed, may be very low. For example, if we assume $P(packed|malware) = 0.92$, $P(packed|benign) = 0.01$, and $P(benign) = 0.98$ (i.e., we assume that 2% of the binaries downloaded by a P2P or web crawler, for example, are malware), using Bayes' formula we obtain $P(malware|packed) \simeq 0.65$. To the best of our knowledge, the real values of $P(packed|benign)$ and $P(benign)$ (and even $P(packed|malware)$ itself) are unknown and may vary based on the approach used for collecting the executables. Therefore, no reliable estimation of the real $P(malware|packed)$ exists (thus, we cannot just assume it is high). This is why McBoost provides two classification steps, namely modules **A** and **C** in Figure 1, instead of just relying on the detection of packed executables (in [19] we also show that signature-based packer detectors suffer from many false negatives, whereas statistical classifiers have much better generalization ability).

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.) obtaining a classification accuracy of 83.4% and an Area Under the ROC curve (AUC) equal to 0.973. As we will discuss in detail in Section 4.1.3, the AUC can be interpreted as an estimate of the probability that the classifier will rank a randomly chosen malware higher than a randomly chosen benign executable (i.e., the probability that McBoost will output a higher $P(malware|e)$ for malware executables than the $P(malware|e)$ estimated for benign executables). Because McBoost is intended to be

used mainly for prioritizing the (possibly manual) detailed analysis of suspicious executables, the AUC is a better evaluation metrics than accuracy for the performance of McBoost. Given that the maximum value for the AUC is 1 and McBoost has an estimated AUC equal to 0.973, our classification approach is very promising. Furthermore, the total time it takes for modules **A** and **C** to classify an PE executable is around 1.06 seconds, on average. Since we expect the percentage of *zero-day* malware to be fairly small compared to the benign (note that here we assume the known malware have already been filtered out using signature-based AV-software), and given that most benign executables are non-packed, the majority of the collected executables will pass directly from module **A** to module **C** and be quickly and accurately classified. Therefore, McBoost provides a fast way to filter out most of the benign executables (because they will be assigned a low $P(malware|e)$), 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.

The remainder of the paper is organized as follows. We discuss the most relevant related work in Section 2 and present the details of our McBoost classification system in Section 3. In Section 4 we report the experimental results, while in Section 5 we discuss the limitations of our approach, possible evasion techniques and countermeasures. We then briefly conclude in Section 6.

2 Related Work

In [23], Royal et al. proposed Polyunpack, an universal unpacker based on a combination of static and dynamic binary analysis. Given a packed PE (portable executable format) executable, a static view of the code is constructed first. Then, the executable is run in a protected environment (a virtual machine). 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 disassemble the hidden code in memory until the disassembling encounters a failure. At that point Polyunpack will dump the code that was successfully disassembled and will try to reconstruct the original executable (i.e., the executable that underwent the packing process). Royal et al. also showed that the detection rate of signature-based AV-software measured on the extracted hidden code using Polyunpack significantly increases, compared to the detection of the packed version of the same malware [23]. However, the main shortcoming of Polyunpack is that it is not able to effectively extract the hidden code of executables that use multiple layers of packing.

Renovo, a different and somewhat more effective tool for universal unpacking using dynamic binary analysis, was presented by Kang et al. in [11]. Renovo runs the executable in an emulated environment and keeps trace of any memory location on which the application writes during execution. Whenever the application jumps to a new instruction, Renovo checks if the location in memory of the instruction to be executed was previously written in any way by the application itself. If this is the case, the executable is in the middle of (a layer of) packing. At this point Renovo will reset the list of memory writes and restart keeping trace of the memory locations that are written during execution. If the application jumps to another instruction that was previously written in memory, Renovo recognizes this event as the end of a layer of packing and the start of the next one [11]. Renovo continues tracking layers of packing until a timeout is reached.

In [16] Martignoni et al. present OmniUnpack, a Windows kernel driver that monitors the execution of applications in memory and detects attempts of executing dynamically decrypted code. If such code is detected, OmniUnpack will scan it using signature-based anti-virus software. Then, if any malicious code is found the execution of the application will be stopped, otherwise it will continue until another attempt to execute decrypted code is detected.

Lyda et al. presented Bintropy [15], a tool for the detection of packed executables based on byte entropy analysis. Bintropy divides the PE file into blocks of 256 bytes. It then computes the entropy of each block,

the average, and the maximum block entropy. Given a dataset of packed binaries, the authors use statistical inference to compute a 99.99% confidence interval on the value of the average block entropy and maximum block entropy [15]. Based on the lower bound of these two confidence intervals, they set two thresholds, one for the value of the average block entropy, and one for the maximum block entropy. During test, if a PE file is found to have average and maximum entropy above the respective thresholds, it will be classified as *packed*. Our work is different in that we do not limit the analysis to the PE file entropy. In [19] we introduce and motivate the use of additional features that help in distinguishing between *packed* and *non-packed* executables, and in this paper we implement a multiple classifier system that combines the approach in [19] with n-gram analysis to detect packed executables more accurately and reliably. Furthermore, in this paper we do not limit our analysis to deciding if an executable is packed or not, but we go further and we classify the hidden code (if any) extracted from packed executables giving in output the probability that the executable under analysis contains malicious code.

In [24] 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) [24]. Three different classification algorithms are used, namely RIPPER (a rule learner), Naive Bayes, and a classifier ensemble called Multi-Naive Bayes. Our work is different from [24] because we adopt a different approach. We first distinguish between *packed* and *non-packed* executables. If an executable is deemed packed, we extract and classify the hidden code into *malware* or *benign*. On the other hand, if the executable is deemed *non-packed* we classify the non-packed code extracted from the executable and detect whether it is malicious or not. Also, we measure a different set of features extracted from PE executables, compared to [24].

To the best of our knowledge, the work closest to ours is [12]. Kolter et al. [12] use n-gram (i.e., sequences of n bytes) 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 [12]. 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 [5, 28], not taking this into account during the training and test of the classifier may produce over-optimistic results. We speculate that 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. Although we use n-gram analysis in our McBoost, our approach is significantly different from [12]. 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 [11]. 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*.

Wang et al. proposed n-gram analysis for payload-based network anomaly detection, and implemented their ideas in PAYL [30] and ANAGRAM [31]. PAYL measures the frequency distribution of n-grams extracted from the payload of normal packets to create a model of normal traffic. During test, each packet is inspected and the distribution of n-grams in the payload is compared to the model of normal traffic. If the distribution of n-grams deviates from the model of normal traffic beyond a predefined threshold, the packet will be flagged as anomalous [30]. ANAGRAM uses a different approach. It stores the distinct n-grams extracted from normal packets in a Bloom filter b_1, and the n-grams extracted from known attacks in a second Bloom filter b_2. During the test phase, for each packet all the distinct n-grams are extracted from the payload and compared with the filters b_1 and b_2. Payloads that contain too many n-grams that are not present in b_1 or that are present in b_2 are classified as anomalous. Li et al. proposed to use *fileprints* [14] to determine

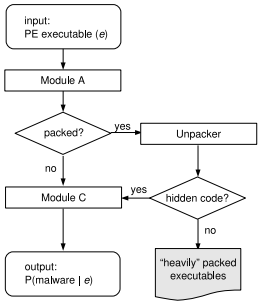

Figure 3: McBoost's decision process.

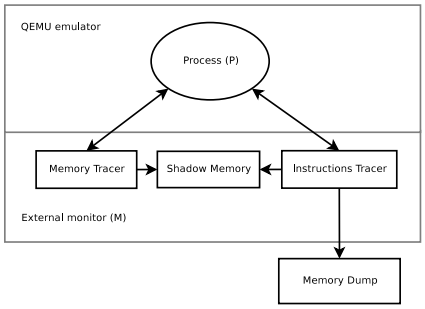

Figure 4: Universal unpacker based on the QEMU emulator.

the type of a file (e.g., PDF, EXE, ZIP, etc.) using statistical classification techniques based on n-gram analysis, and in [29] Wang at al. experimented on using fileprints for the detection of malicious code hidden in normal documents or applications. Our work is different because we do not focus on the classification of anomalous network traffic, or on the detection of malware embedded in normal documents. We use n-gram analysis and combine it to text classification techniques and other heuristics-based approaches for the detection of malicious PE executables.

3 McBoost

McBoost consists of three modules (see Fig. 1): **A**) a classifier that detects if an executable is packed or not; **B**) a dynamic universal unpacker; and **C**), a binary code classifier that detects whether the executable's (non-packed or unpacked) code is malicious. Modules **A** and **C** are further subdivided in sub-modules. As depicted in Fig. 3, given an executable e, if module **A** decides that e is packed it will be sent to the unpacker, otherwise e's code section will be sent to **C** (in particular to **C1**). Assuming e is deemed packed by **A**, the unpacker has the responsibility of extracting the hidden code from e and passing it to **C**. There are two possibilities. If the unpacker was able to extract the hidden code, the hidden code will be passed to **C** (in particular to **C2**), which is the module specialized in detecting if the hidden code is malicious or not. On the other hand, if the unpacker was not able to extract any hidden code (perhaps because e implements strong anti-emulation techniques), e's code section 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.). For the executables that are sent to module **C** (either to **C1** or **C2**) McBoost will output an estimate of the probability $P(malware|e)$ of e being malicious.

3.1 Detecting Packed Executables

Module **A** performs detection of packed executables using a Multiple Classifier System (MCS) [13] 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**). 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**). The approach we use for module **A1** is borrowed from [19][1], where we show that our heuristics-based classifier has much

[1][19] has been submitted by the authors to the Pattern Recognition Letters Journal (`http://www.elsevier.com/locate/pr`) in October 2007, and is currently under review. We'd like to emphasize that the study presented in [19] was focused only on the classification of packed vs. unpacked executables and was restricted to the heuristics-based classifier (module **A1**). In this work we propose to use the heuristics-based classifier in combination with two other classifiers based on n-gram analysis and text classification techniques for making the detection of packed executables more accurate and ro-

higher accuracy than signature-based approaches like PEiD (`http://www.peid.info`), for example. Here we combine module **A1** to **A2** and **A3** in order to make the classification of packed executables even more accurate and robust against evasion (we discuss possible evasion techniques and countermeasures in Section 5).

3.1.1 Heuristics-based Classifier

Module **A1** is a Multi-Layer Perceptron (MLP) classifier trained in order to distinguish between packed and non-packed PE executables. The choice of the Multi-Layer Perceptron is justified by the very good results we obtained in a previous study [19], where we show that our heuristic-based MLP classifier has a much better accuracy and generalization ability (i.e., less false negatives) for detecting packed executables compared to PEiD. Furthermore, MLP classifiers usually produce well-calibrated posterior class probabilities [17], which may be useful when combining multiple classifiers (see Section 3.1.3). The details of how the classifier is trained are reported in Section 4. Here we describe how a PE file is translated into a pattern vector suitable as input for a statistical classifier. We measure nine features that were derived from some heuristics for detecting packed executables based on our experience, whereby each PE executable is represented as a vector $x \in \mathbb{R}^9$ [19]:

Number of *Standard* and *Non-Standard* Sections. The PE file of non-packed applications usually contains a well defined set of standard sections. For example, applications compiled using Microsoft Visual C++ usually contain at least one code section named `.text`, and two data sections named `.data`, and `.rsrc` [20][2]. On the other hand, packed executables often do not follow standard section names. For example, the UPX executable packing tool (`http://upx.sourceforge.net`) usually creates PE files that contain two sections named `.UPX0` and `.UPX1`, respectively, and a section named `.rsrc`. A number of other executable packing tools usually generate PE files that contain code and data sections having non-standard names. Therefore, counting how many standard and non-standard section names are present in a PE file may indicate weather the executable is packed or not (notice that the number of standard section and the number of non-standard sections are two distinct features).

Number of Executable Sections. When analyzing the output of executable packing tools, we noticed that the PE file of some packed executables do not carry any executable section. This is obviously anomalous, because if the operating system does not allow the execution of instructions in data sections the application will crash, for example, in Windows XP Service Pack 2 [1]. However, packed executables that do not contain any executable sections may still run correctly in older versions of Windows. On the other hand, the `.text` section (i.e., the standard code section) of non-packed executables is always correctly marked as executable[3]. Therefore counting the number of executable sections in the PE file may help in distinguishing between packed and non-packed executables.

Number of Readable/Writable/Executable Sections. By definition, packed executables need to unpack the hidden code and copy it in memory so that it can be executed. Therefore, the PE file of a packed executable needs to include at least one section that is Readable/Writable/Executable at the same time. On

bust (see Section 3.1.3). Furthermore, our McBoost classification system is not limited to the detection of packed vs. non-packed executables, but is intended for fast classification and ranking of malware vs. benign executables. We believe the overlap between this paper and [19] is therefore very limited. A confidential copy of [19] is available for the reviewers at `http://roberto.perdisci.googlepages.com/prl-submission`.

[2]The standard section names reported in [20] are related to Microsoft compilers. We also consider as standard the section names generated by Borland compilers.

[3]This is true unless a "non standard" compiler is used to generate the PE file.

the other hand, the executable sections in the PE file of non-packed applications do not need to be writable, and the Writable section flag usually is not set. Therefore, counting the number of sections which are writable and executable at the same time adds a piece of evidence to the conclusion whether the executable is packed or not.

Number of Entries in the IAT. The Import Address Table (IAT) of an application contains the address of the external functions called by the application. These external functions are typically imported from Dynamically Linked Libraries (DLL). Most non-packed executables import many external functions. For example, they usually import many functions from the native Windows API, which are used to read/write files, open new windows on the screen, manage a network connection, and so on. Therefore, the IAT will usually contain many entries, one per each imported function. On the other hand, packed executables often import very few external functions. The main reason is that the unpacking routine does not need many external functions. The basic operations the unpacking routine performs are reading and writing memory locations in order to decrypt the code of the packed application on the fly. For example, no window on the screen or network operation is usually needed. This is reflected in a small number of entries in the IAT of a packed executable (the original (packed) application will likely need to perform operations on the screen and network. Therefore, unpacking routines often overwrite the IAT in memory on the fly to add the missing IAT entries).

Code and Data Entropy. The hidden code of a packed application is usually stored in a portion of the code or data sections of the packed PE file (we identify a section as a code section if the Executable section flag is set, otherwise we consider the section to be a data section). Since the hidden code is usually encrypted, it will look "random". On the other hand, non encrypted code sections contain well-structured information, namely the opcode of executable instructions and the memory location of the operands. Non-encrypted data sections also contain somehow structured information. Following this observation, we measure the byte entropy of the code and data sections of the PE file. If the entropy of a section is close to 8, which is the maximum byte entropy, the section likely contains encrypted code.

PE Header and Overall File Entropy. The code and data sections are not the only places where the executable packing tool may hide the code of the original application. There are parts of the PE header dedicated to optional fields that are not necessary for the correct loading of the program into memory by the operating system. Some packing tools may therefore hide encrypted code in those unused portions of the PE header. For this reason we measure the byte entropy of the PE header as well. Considering that the PE file is quite complex and contains other such unused spaces (for example, portions of the header of each section), the encrypted code may be hidden in several other locations [15]. Therefore, we also measure the entropy of the PE file as a whole to take into account these cases.

3.1.2 n-gram-based Classifiers

Modules **A2** and **A3** perform n-gram analysis of the code section and of the entire PE file, respectively. For both the classifiers we use a text categorization approach similar to [12]. 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 dis-

criminative (or "powerful") features that allow us to separate instances of the positive and negative class, according to an information gain metric [33]. Specifically, the information gain of an n-gram g is

$$IG(g) = -\sum_{c\in\{\mathcal{P},\mathcal{N}\}} P(c)\log P(c) + \sum_{c\in\{\mathcal{P},\mathcal{N}\}} P(c|g)\log P(c|g) + \sum_{c\in\{\mathcal{P},\mathcal{N}\}} P(c|\bar{g})\log P(c|\bar{g}) \quad (1)$$

where c 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 [4] for both **A2** and **A3**. Bagged-Decision-Trees usually perform very well and have the characteristic of producing calibrated posterior class probabilities [17], which may be useful when combining multiple classifiers (see Section 3.1.3). The details regarding the algorithm and parameters we used for training the classifiers are reported in Section 4.

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 $n \geq 3$, 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.

3.1.3 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) [13] 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) [13]. Diversity may be induced in a number of ways [7]. 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 [7]. We chose Multi-Layer Perceptron and Bagged-Decision-Tree as base classifiers

because they both output well-calibrated posterior class probabilities [17]. Also, MCS are more robust and have been shown to be useful in making evasion by mimicry attacks harder [18] (we discuss possible evasion techniques against McBoost and countermeasures in Section 5).

We use a simple but effective combination rule to combine the output of **A1**, **A2**, and **A3**, namely the average of probabilities

$$P_A(\mathcal{P}|e) = \frac{1}{3} \sum_{i=1..3} P_{A_i}(\mathcal{P}|e) \tag{2}$$

and we set a decision threshold θ so that if $P_A(\mathcal{P}|e) > \theta$ 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**.

3.2 Extracting Hidden Code

In this section we describe how our universal unpacker (module **B**) works. Our unpacker is based on a dynamic binary analysis approach, and is basically an implementation of Renovo [11], which we adapted in order to make it work in McBoost (because Renovo is not publicly available, we implemented our own version according to the description reported in [11]).

We built the unpacker on top of the QEMU emulator [2] (see Fig. 4). In essence, we slightly modified QEMU to introduce a plug-in functionality. Then, we implemented our unpacker as a plug-in of QEMU. An external monitor M (i.e., external to QEMU) receives call-backs from the emulator, and is then able to follow the execution of a process P and trace its memory operations. M uses a shadow memory (basically a *Set* data structure) in order to keep trace of every memory address written by P. Every time an instruction i is executed on behalf of P, M will intercept it and query the shadow memory to check whether it contains the address of i. If so, this means that the instruction i was previously dynamically generated by P itself, and will be marked by M as *hidden* code. i will then be considered as the first instruction of a *layer* of unpacking [11]. At this point, M will reset the shadow memory and start tracing P's memory operations, again. Whenever M 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. M will then dump the binary code of the first layer to disk and will keep tracing the execution of the next layer [11].

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 M 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 M keeps trace of the instructions in the QEMU translation blocks [2] (or basic blocks) that were actually executed by the emulator (our unpacker is different from Renovo [11] in this case. To the best of our knowledge Renovo only implements an approach very similar to the *bpage* dumps described above).

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.

Since determining whether an executable is packed or not is undecidable [23], there is no reliable way to say if and when a process P has completed the unpacking process. Therefore, we use a heuristic approach to decide when to terminate the execution of P. We set two timers, namely a *per layer* timer and a *global*

timer. The former is reset whenever a new layer is detected, whereas the latter starts at the beginning of the execution of P. If the *per layer* timer reaches a *per layer* time-out T_l before a new layer is detected, M will terminate P and dump the last layer of unpacking. On the other hand, if the *per layer* time-out is never reached, M will keep executing P until the *global* timer reaches a *global* time-out T_g. Apart from reaching a time-out, there are other reasons why M 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 (e.g., if we try to run a non-Win32 application, or an executable with a corrupted PE header).

3.3 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. Here again, the intuition behind the use of n-gram analysis is that the presence or absence of an informative n-gram (see Section 3.1.2) 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 prevalent instructions are used to carry out malicious activities and may not be present with the same frequency or sequence in benign code. Therefore, modules **C1** and **C2** try to capture this difference in the distribution of n-grams between malicious and benign code.

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 4.1.1), 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 3.1.2, 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 3) and shown in Fig. 1.

4 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.

4.1 Experimental Setup

4.1.1 Preparation of the Datasets

To evaluate the effectiveness of McBoost, we collected several thousands of Windows benign and malicious PE executables. 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 [23] 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[4] [11] in order to further filter any possibly packed executable missed (i.e., no hidden code was detected) by Polyunpack and our unpacker. 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 (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.

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

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.

4.1.2 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 3) 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 (i.e., the number of features), whereas the output layer has one node (because we are dealing with a two-class problem). We set the number of perceptrons in the hidden layer to 7. Modules **A2**, **A3**, **C1**, and **C2** (see Section 3) 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. 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 3.2).

4.1.3 Validation Metrics

We validate our classification system using the accuracy, Receiving Operating Characteristic (ROC) curve analysis, and the Area Under the ROC (AUC). We now describe how these metrics are computed, and explain their meanings.

Accuracy. The accuracy of a classifier is the fraction of correctly classified instances. Given a labeled test dataset $\mathcal{T} = (e_i, l_i)_{i=1..N}$, where e_i are the instances to be classified (i.e., PE executables in our case) and $l_i \in \{\mathbf{p}, \mathbf{n}\}$ are their true labels ($\mathbf{p}$ stands for positive, whereas $\mathbf{n}$ stands for negative), the accuracy is computed as

$$Accuracy = 1 - \frac{1}{N} \sum_{i=1..N} L(l_i, \hat{f}(e_i)) \tag{3}$$

where $\hat{f}(e_i) \in \{\mathbf{p}, \mathbf{n}\}$, and $L(l_i, \hat{f}(e_i))$ is the 0/1 loss function (i.e., $L = 0$ iff $l_i = \hat{f}(e_i)$, and $L = 1$ otherwise).

In our case, all the classifiers in McBoost, and the overall output of McBoost itself, can be interpreted as a posterior class probability $P(l = \mathbf{p}|e)$ (see Section 3), where $\mathbf{p}$ is the *positive* class label (e.g., *packed*, or *malware*). In order to translate this output into a label and compute the accuracy, we set a threshold θ and say that e is of class $\mathbf{p}$ if $P(l = \mathbf{p}|e) > \theta$, otherwise we say that e is of class $\mathbf{n}$.

Although accuracy is one of the most common metrics used to evaluate the performance of a classifier, it suffers from two major problems: it depends on the value of the threshold θ and on the *a priori* class probabilities, i.e., the proportion of positive and negative instances in the test dataset [3]. For example, assume $\mathcal{T}$ contains 99% of negative instances and 1% of positive instances, i.e., the *a priori* class probabilities are $P(\mathbf{n}) = 0.99$ and $P(\mathbf{p}) = 0.01$, respectively. If the classifier always outputs $\hat{f}(e) = \mathbf{n}$, regardless of the input e, the accuracy will be 99%, although the classifier is unable to detect any instance of the positive class.

ROC analysis. By varying the decision threshold θ on the probability $P(l = \mathbf{p}|e)$ that an instance e (an executable in our case) belongs to the positive class $\mathbf{p}$ (e.g., *packed* in case of the output of module **A**, or *malware* in case of module **C**), we can vary the trade-off between the false positive rate (FP) and the detection rate (DR) of a classifier. The false positive rate represents the fraction of negative instances (e.g., *non-packed* or *benign*) classified as positive (e.g., *packed* or *malware*). On the other hand, the detection

Classifier	Accuracy	False Positive Rate	Detection Rate	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 (threshold = 0.5).

rate represents the fraction of positive instances correctly classified as positive. The Receiver Operating Characteristic (ROC) curve shows how the FP/DR trade-off varies while varying the detection threshold.

AUC. The AUC is the area under the ROC curve. Intuitively, the higher the AUC the better, given that we can find (in general) a better trade-off between false positives and detection rate.

The AUC can be computed as follows. Let $\mathcal{T} = (e_i, l_i)_{i=1..N}$ be a test dataset, where e_i are the instances to be classified (i.e., a PE executable in our case) and l_i are their true labels. Suppose again that the labels l_i can assume only two values $l_i \in \{\mathbf{p}, \mathbf{n}\}$, where $\mathbf{p}$ stands for positive and $\mathbf{n}$ stands for negative, and let $\mathcal{T}^{(\mathbf{p})}$ be the subset of $\mathcal{T}$ consisting of all the instances of the positive class, and $\mathcal{T}^{(\mathbf{n})}$ be the subset of $\mathcal{T}$ consisting of all the instances of the negative class. Assume also that given an instance e_i the classifier outputs the posterior probability for the positive class $P(l_i = \mathbf{p}|e_i)$. In this case, the Area Under the ROC Curve (AUC) can be estimated using the Wilcoxon-Mann-Whitney statistic [6]

$$AUC = \frac{\sum_{j=1..m} \sum_{k=1..n} I(x_j > y_k)}{m \cdot n} \tag{4}$$

where $x_j = P(l_j = \mathbf{p}|e_j), \forall e_j \in \mathcal{T}^{(\mathbf{p})}$, $y_k = P(l_k = \mathbf{p}|e_k), \forall e_k \in \mathcal{T}^{(\mathbf{n})}$, and $I()$ is the indicator function whereby $I(x > y) = 1 \iff x > y$, otherwise $I(x > y) = 0$.

The AUC ranges from 0 to 1 and can be interpreted as an estimate of the probability that the classifier will rank a randomly chosen instance of the positive class $(e_i, l_i = \mathbf{p})$ higher than a randomly chosen instance of the negative class $(e_j, l_j = \mathbf{n})$ [6]. Also, the AUC has several advantages compared to the accuracy [3] for evaluating the performance of a classifier. In particular, it is independent from the *a priori* class probabilities (i.e., the proportion of positive and negative instances in the test dataset), and is independent from the decision threshold θ [3].

4.2 Validation of Single Modules

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.

Module B. In order to validate the performance of 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

Classifier	Accuracy	False Positive Rate	Detection Rate	AUC
C1 (non-packed code)	0.963	0.026	0.750	0.927
C2 *bpage* (hidden code)	0.878	0.111	0.877	0.932
C2 *bbexec* (hidden code)	0.865	0.139	0.865	0.909

Table 2: Validation of module **C1** and **C2** for *malware* vs. *benign* code classification (threshold = 0.5).

Classifier	Accuracy	False Positive Rate	Detection Rate	AUC
McBoost	0.834	0.025	0.810	0.973

Table 3: Validation of McBoost (threshold = 0.5).

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.

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 3.2). 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 3.2), instead of the *bpage* dumps. It is easy to see that using *bpage* dumps provides better results, 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.

4.3 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

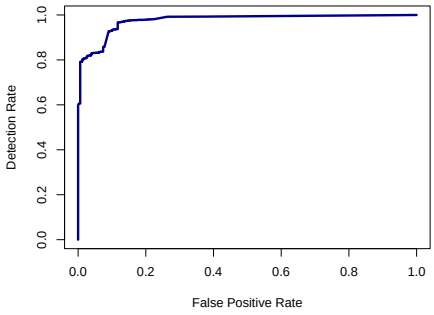

Figure 5: Receiver Operating Characteristic (ROC) curve for McBoost.

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 where 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 (we set the detection threshold equal to 0.5). Figure 5 shows the ROC curve, whereas Table 4 reports the values of false positives, detection rate and detection threshold for significant points on the ROC curve. As we can see, McBoost has an estimated accuracy of 83.4% (for a threshold equal to 0.5), and an Area Under the ROC curve (AUC) equal to 0.973. It is also worth noting that if we set the detection threshold to 0.912, McBoost is still able to correctly detect 60% 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 0.01 and obtain 99.1% detection rate with 26.1% of false positives.

We would like to emphasize that the most important result is the value of the AUC (which is equal to 0.973 in our experiments), because the AUC can be interpreted as the probability of ranking a malware executable higher than a benign executable, as explained in Section 4.1.3. 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 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.

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

Threshold	False Positive Rate	Detection Rate
0.912	0	0.6
0.722	0.006	0.791
0.500	0.025	0.810
0.293	0.047	0.831
0.143	0.099	0.931
0.010	0.261	0.991

Table 4: Significant values of the trade-off between false positives and detection rate for McBoost.

mostly benign executables and a small fraction of unknown malware. Also, we expect most of the benign to be non-packed (in Section 4.1.1 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, the time saving due to McBoost was about 43.8%. 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 the time saving due to McBoost to be around 86.6%.

5 Limitations and Future Work

There are some limitations to McBoost. The first limitation is in the fact that McBoost relies in part on the results of the dynamic-analysis-based unpacker. As advanced executable packing tools may implement sophisticated emulator and virtual machine detectors [22], our unpacker may not be able to extract the hidden code from the executables packed using such sophisticated techniques. Therefore, the last classification step (module **C**) may not be utilized and, if this is the case, McBoost can only say that the executable is (likely) "heavily" packed (see Section 3). This problem may be solved by implementing the unpacker so that it will recognize any attempt to detect the emulated or virtual machine environment, and *lie* back to the executable that is checking for their presence. Our unpacker implements some of this countermeasures in a way similar to Renovo [11]. Improvements to the anti-evasion techniques of our dynamic unpacker will be the subject of future work.

Some executable packing tools, for example Themida (`http://www.oreans.com/themida.php`), are able to transform blocks of the original executable code into a proprietary byte-code version that runs in a virtual machine environment internal to the packed executable. In this case, even if the unpacker was able to extract the hidden code, this hidden code will likely be a proprietary byte-code representation of the original code. Therefore, it is difficult for module **C2** (the classifier responsible for distinguishing between malware vs. benign hidden code) to make a correct decision. A possible countermeasure would be to "force" the classifier **C2** to learn to detect such byte-code as malicious (by giving it suitable samples during the training phase). As a consequence, any executable (malware or benign) packed with Themida, for example, will undergo detailed (possibly manual) analysis. Because most of the benign are non-packed, and (we suspect) the percentage of packed benign actually packed with Themida (or similar sophisticated packers) is likely low, McBoost may generate few false positives, but it will avoid a possibly large number of false negatives.

Another limitation comes from the fact that a malware writer may try to evade detection by McBoost. For example, in case of the heuristics-based classifier (see Section 3.1.1), an attacker may try to evade it by packing a malware executable with a modified packing tool. In this case the attacker may be able to easily control things like the number of standard and non-standard section names, the entries in the IAT, where the hidden code is stored, etc. Also, the attacker may try to modify the entropy of the header, code, and data sections, and of the entire file, in order to decrease their value and make the PE executable look like non-packed. For example, in case of the code section, the attacker may inject a certain amount of "normal looking" junk code in order to decrease the entropy of the section. Similarly, the attacker may inject junk data into the data sections to try and decrease their entropy. These techniques may also be used to evade the n-gram based classifiers (see Section 3.1.2).

However, there are parts of an executable/malware that may not be modified so easily. For example, the presence of a Readable/Writable/Executable section may be necessary for the unpacking procedure to work properly (see Section 3.1.1). Also, it is not clear how much junk code (or data) the attacker needs to add in order to evade multiple entropy and n-gram based classifiers at the same time. Adding too much junk code (or data) to the packed executable may cause it to grow significantly in size, thus making the propagation of the malware slower, or raising the suspicions of an intrusion detection system. Furthermore, it may be possible to implement effective countermeasures to such evasion attempts. For example, instead of computing the entropy on an entire section (or file), we could split it in a random number of blocks, each one starting at a random offset, and measure the maximum block entropy, instead of the average entropy. This way, the blocks that "cover" (portions of) the hidden/encrypted code will still cause the entropy used for classification to have a high value. This technique may be also adapted and used for the n-gram classifiers (see Section 3.1.2). In this case the attacker would need to guess multiple locations where the entropy and the most informative n-grams will be measured and try to add junk code or data in these guessed locations, or try to interleave the junk code and data with the unpacking routine and the hidden code. We believe such kind of attacks are hard to devise.

Analogous attack scenarios and countermeasures have been proposed in [9, 8, 31] in the context of payload-based anomaly detection. In [8], Fogla et al. presented a formal framework for constructing polymorphic blending attacks (PBA). PBA are mimicry attacks that transform the payload of attack packets in order to match the distribution of n-grams (and possibly other features) in normal traffic, while maintaining the ability to exploit the targeted vulnerability. Fogla et. al [9, 8] formally proved that such kind of attacks are NP-complete in the general case, and showed experimentally that it may be possible to find approximate solutions that effectively evade PAYL [30]. In order to cope with PBA, Wang et al. proposed ANAGRAM [31] and introduced the concept of *randomized n-gram models*, which raise the bar with respect to mimicry attacks against payload-based anomaly detectors based on n-gram analysis. These studies are very relevant to our work on possible evasion techniques against McBoost and the corresponding countermeasures.

The evaluation of the effectiveness of the attacks and countermeasures we discussed will be the subject

of our future work.

6 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 quickly filter out most of the benign and prioritize further detailed analysis of the remaining suspicious binaries. For example, McBoost may be used for the analysis of binaries downloaded through a P2P (or web) *executable crawler* searching for previously unknown (*zero-day*) malware from which to extract a detection signature. In this case, after filtering known malware using existing AV-software, we expect the collected set of executables will contain mostly benign applications and a small fraction of unknown malware. We showed that our system is able to classify most benign executables in around 1.06 seconds per binary, and that it provides a fast way to filter out benign executables and prioritize the analysis of the most 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 83.4% and an AUC equal to 0.973. In addition, the running time of McBoost on our test data shows a significant time saving (e.g., 86.6%) over existing analysis approaches.

References

[1] S. Andersen. Changes to functionality in Microsoft Windows XP Service Pack 2, part 3: Memory protection technologies. `http://technet.microsoft.com/en-us/library/bb457155(d=printer).aspx`.

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

[3] A. P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7):1145–1159, 1997.

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

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

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

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

[8] P. Fogla and W. Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In *ACM Conference on Computer and Communications Security*, 2006.

[9] Prahlad Fogla, Monirul Sharif, Roberto Perdisci, Oleg Kolesnikov, and Wenke Lee. Polymorphic blending attacks. In *USENIX Security Symposium*, 2006.

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

[11] 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.

[12] 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.

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

[14] W. Li, K. Wang, S. J. Stolfo, and B. Herzog. Fileprints: Identifying file types by n-gram analysis. In *IEEE Workshop on Information Assurance*, 2005.

[15] R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security and Privacy*, 5(2):40–45, 2007.

[16] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Annual Computer Security Applications Conference (ACSAC)*, 2007.

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

[18] 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.

[19] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. Classification of packed executables for accurate computer virus detection, October 2007. Submitted to Patter Recognition Letters Journal. Currently under review. A confidential copy is available for the reviewers at `http://roberto.perdisci.googlepages.com/prl-submission`.

[20] M. Pietrek. An in-depth look into the Win32 Portable Executable file format, part 2. `http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/`.

[21] 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.

[22] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting system emulators. In *Information Security Conference (ISC)*, 2007.

[23] 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.

[24] 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.

[25] Heise Security. Antivirus protection worse than a year ago, December 2007. `http://www.heise-security.co.uk/news/100900`.

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

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

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

[29] S. J. Stolfo, K. Wang, and W. Li. Fileprint analysis for malware detection. In *Technical Report, Columbia University, NY*, June 2005.

[30] K. Wang and S. Stolfo. Anomalous payload-based network intrusion detection. In *Recent Advances in Intrusion Detection (RAID)*, 2004.

[31] K. Wang and S. Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In *Recent Advances in Intrusion Detection (RAID)*, 2006.

[32] 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.

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

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