Sādhanā Vol. 40, Part 5, August 2015, pp. 1435–1456. © Indian Academy of Sciences

An efficient block-discriminant identification of packed malware

SMITA NAVAL[1,*], VIJAY LAXMI[1], MANOJ SINGH GAUR[1] and VINOD P[2]

[1]Department of Computer and Science Engineering, Malaviya National Institute of Technology, Jaipur 302 017, India
[2]SCMS School of Engineering, Kerala 683 582, India
e-mail: smita.710@gmail.com; vlaxmi@mnit.ac.in; gaurms@mnit.ac.in; pvinod21@gmail.com

MS received 28 July 2014; revised 26 December 2014; accepted 24 April 2015

Abstract. Advanced persistent attacks, incorporated by sophisticated malware, are on the rise against hosts, user applications and utility software. Modern malware hide their malicious payload by applying *packing* mechanism. Packing tools instigate code encryption to protect the original malicious payload. Packing is employed in tandem with code obfuscation/encryption/compression to create malware variants. Despite being just a variant of known malware, the packed malware invalidates the traditional signature based malware detection as packing tools create an envelope of packer code around the original base malware. Therefore, unpacking becomes a mandatory phase prior to anti-virus scanning for identifying the known malware hidden behind packing layers. Existing techniques of unpacking solutions increase execution overhead of AV scanners in terms of time. This paper illustrates an easy to use approach which works in two phases to reduce this overhead. The first phase (ESCAPE) discriminates the packed code from the native code (non-packed) by using random block entropy. The second phase (PEAL) validates inferences of ESCAPE by employing bi-classification (packed vs native) model using relevant hex byte features extracted blockwise. The proposed approach is able to shrink the overall execution time of AV scanners by filtering out native samples and avoiding excessive unpacking overhead. Our method has been evaluated against a set consisting of real packed instances of malware and benign programs.

Keywords. Malware; obfuscation; packing; unpacking; entropy; classification algorithm.

*For correspondence

1. Introduction

Malware is a common channel by which crackers (the bad guys) facilitate cybercrime. It has become a weapon whose use intersects multiple threats confronted by security researchers across the globe (Dinaburg *et al* 2008). The crackers keep on enhancing the complexity of malcode to accomplish their notorious intentions. They use a variety of obfuscation techniques to foil researchers and operate as covertly as possible on a user's system (Kevin 2010). According to Symantec threat report 2014,[1] the volume and sophistication of malicious activities have increased substantially. These trends are due to widespread use of packing technology as it provides a mechanism to mask the actual functionality of malicious binaries and avoid detection (Christopher 2010). Survival from detection is the main motive of malware writers. Packed malware compromises the integrity of our system by hiding the malicious contents of an executable. Packing methods transform some or all of the original bytes into a series of random–looking data bytes that appear in 80–90% of malware samples (Lyda & Hamrock 2007; Brosch & Morgenstern 2006). Such hidden malware enter into our system and stealthily achieve various malicious objectives like installing keyloggers, modifying file and directory structures, and disabling AV scanners and firewalls. Packing tools are freely and commercially available. Packers provide malware authors a door of opportunity to infect the user's computer using encryption or compression algorithms for their financial benefits. To detect and analyze the infection characteristics of packed malware, unpacking becomes an important phase prior to malware detection and analysis.

To deal with packed malware, AV-scanners employ unpacking mechanism to draw a fair conclusion about the binary under consideration. Numerous unpacking solutions are available which incorporate static and dynamic unpacking techniques. State-of-the-art malware detectors have adopted both static and dynamic techniques to recover the payload of packed malware, but unfortunately such techniques are highly ineffective (Martignoni *et al* 2007). These unpacking approaches provide solution at the cost of increased time-complexity (Bohne 2008; Kang *et al* 2007; Martignoni *et al* 2007; Royal *et al* 2006; Vilkeliskis 2009). Each packed executable must undergo unpacking phase to ensure correct scanning. For instance, the static unpacking methods (Coogan *et al* 2009; Jeong *et al* 2010) depend on the collection of previously known packing algorithms and their counterparts *i.e.*, unpacking algorithms. These static unpackers unpack the packed malware by first identifying the packer applied and then applying unpacking routine on the packed sample. In this way, static unpacker extracts the original payload. The static unpacker's dependency on packer database makes this approach ineffective with unknown packers. If a malware binary is packed with known packer, such as UPX and ASPACK, then the static unpacking methods can be applied. However, the malware authors can evade these methods by slightly modifying the packer algorithm or can use their own encryption or decryption algorithms (Coogan *et al* 2009). Also, the new version of previously known packers increases the size of packer database and thus enhances the *time-complexity* of the static unpacking. On the other hand, the dynamic unpackers (Kang *et al* 2007; Royal *et al* 2006) extract the original payload of packed file by executing it into a virtualized or safe environment. During the course of execution the original payload is extracted into memory and those regions of memory area are dumped to retrieve the malware payload. The dynamic unpackers, however require dedicated hardware and software settings for unpacking and dumping the packed malware. These unpacking methods are

[1] Internet Security Threat Report 2014. http://www.symantec.com/security_response/publications/threatreport.jsp

time-consuming and susceptible to conditional execution of unpacking modules. Summarizing, following are the main reasons of unpacking being sluggish (slow processing time).

- Large number of malware samples entering into our computer systems.
- The continuous augmentation in the size of packer signature database (Static Unpacking), as polymorphic packing engines changes the appearance of packer code every time (Vilkeliskis 2009).
- Manual unpacking uses debuggers to unpack a single packed binary with human intervention.
- Extracting original payload by executing packed file in virtual environment requires resources and is prone to time-bomb attacks (Dinaburg *et al* 2008).

The volume of malicious binaries is exponentially increasing day by day. These vicious files are equipped with numerous evasion techniques such as polymorphism, metamorphism, oligomorphism and packing. We, in this approach, address the detection of packed malware only. The proposed mechanism does not provide an unpacking solution but aims to reduce the overhead attached with unpacking engine by lightening its input buffer. It detects if an input sample is packed or not. Our approach can be used by AV-scanners to filter out packed samples that need be processed by unpacking engine and blocks non-packed samples entering into unpacking phase. So, our main objective is to minimize the input samples for the unpacking engine and thus improving the efficiency of AV scanners. To carry out the proposed objective, we prefer static approach over execution-based dynamic approaches. The static solutions are more time and memory efficient as compared to dynamic solutions. The dynamic solutions rely on isolated virtualized environments for executing samples. To create these environments, numerous virtual or sandboxing frameworks are available such as Ether, cuckoo, qemu, temu, anubis, to name a few. These instrumented "virtual" frameworks used to confine malware are slower than the "real" host systems (Jacob *et al* 2013). Executing each sample in these isolated frameworks results into a several minutes of execution overhead. To avoid these execution overheads, researchers also used symbolic execution techniques (Avgerinos *et al* 2014; Saxena *et al* 2009) in which symbolic values are assumed to generalize the normal execution of binaries. The symbols represent the constraints for all possible outcomes of each conditional branch. The symbolic execution technique also suffers from path explosion and program-dependency efficacy. As a result, it is not suitable to large program binaries and also lead to state explosion. Keeping these facts in mind and trying to develop a fast approach for reducing the unpacking overhead of AV scanners, we present a static approach that works at binary level. The proposed approach considers Windows Portable Executable (PE) file format due to its high popularity among computer users as well as among all security breachers (virustotal.com/statistics/). Evaluation of our approach on a dataset consisting of real packed malware and benign samples is indicative of its ability to handle different types of packer. The contributions of proposed work are as follows.

- We present an entropy-based approach (ESCAPE) to distinguish a packed sample from non-packed (native) samples. Entropy as a metric can be used to measure the randomness of a packed malware. As the packing tools randomly distribute the packer code around the malware binary. Underlying hypothesis of our entropy-based approach is that *"in comparison to a native sample, packer code is randomly distributed"*. Due to incurred randomness, the packed sample becomes unstructured and results into higher entropy values.
- We compute block-wise entropy for multiple blocks in a sample. By doing so, we capture intermittent parts of the sample that are affected by packing.

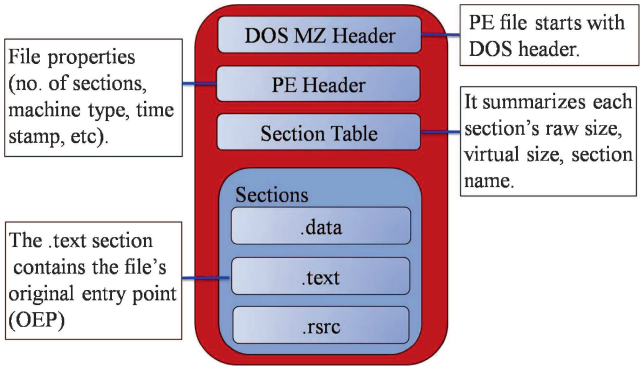


Figure 1. PE file format for Windows.

– We develop a classification model (PEAL) to validate the *heuristic* of entropy-based model. The proposed model is a bi-classification model which differentiates packed and native binaries. For this, feature vector table (FVT) is created using hex byte features extracted blockwise. We apply five classification algorithms to classify packed and native classes. For each learning algorithm, 2, 3, 5, 7, and 10 fold cross-validation is applied.
– We create packed and encoded datasets to evaluate the performance of our models. The packed samples are packed with the packers that do not preserve entropy after packing and the encoded datasets contain samples that are encoded with entropy-preserving algorithms. Our approach ESCAPE works well with packed samples but does not succeed against encoded samples. Our second approach PEAL, on the other hand, can detect packed and encoded samples with higher detection accuracy.

This paper is outlined as follows: Section 2 elaborates the PE file format for Windows binary. Packing and unpacking mechanisms are illustrated in Section 3. Proposed methodologies (ESCAPE and PEAL) for identifying packed and native binaries are introduced in Section 4. Work flow for ESCAPE and PEAL is presented respectively in Section 5 and Section 6. In Section 8, we present and analyze the related work. Finally, concluding remarks are presented in Section 9.

2. Portable executable (PE) format

This section describes PE format used by Microsoft Operating Systems. The name "Portable Executable" refers to the fact that it is not architecture specific.[2] Figure 1 shows components such as various headers and sections that a PE file must contain. These components include section data, import and export tables, resources, etc. (Wei *et al* 2008). PE file starts with DOS MZ header. Windows versions check the compatibility of the file by looking at this header. If a file does not contain MZ header, a warning is displayed with a message "This program cannot be run in DOS mode". The PE header includes basic file properties like number of sections, time stamp and machine type. Section table contains raw size, virtual size and name of each section. After section table, contents of all sections are placed. These sections could be `.data, .code,`

[2]http://msdn.microsoft.com/library/windows/hardware/gg463125

`.bss, .text, .rdata, .debug`. Each PE file contains some relevant standard fields such as `SizeOfCode, SizeOfInitializedData, AddressOfEntryPoint` and `SizeOfUninitializedData`.

3. Packing and unpacking mechanism

Packing is a method in which an executable is encrypted or compressed to protect its original contents. Malware authors employ packing to hide their malicious payload. Applying one or multiple packer layers, original entry point (OEP) is obfuscated to malware detector. To hide OEP, a fake entry point is defined. When executed with this fake entry point, malicious code executes in benign mode bypassing AV detection mechanisms before redirecting to OEP and delivering payload. When these packed files are executed from OEP, the unpacking routine is invoked and malicious payload is loaded into memory. Packed and archived files are different in a way that the former are directly loaded into main memory while the latter are unzipped into secondary memory. This packing mechanism has proven beneficial to malware writers as it fulfills their motive of keeping their malcode invisible to malware detectors. UPX, ASPACK, PECOMPAT, NSPACK, PETITE, exe32Pack are some freely available packing tools used by malware authors to obfuscate their code. In this paper, we use UPX,[3] ASPACK[4] and PECOMPACT,[5] the most popular among malware authors (Kang *et al* 2007) and entropy-preserving XOR-based encoders such as shikata ga nai.[6] These packers are selected for experimentation because these have been mostly employed by malware authors (as reported from virustotal submission virustotal.com).

The effectiveness of malware detectors depends on the ability to recover the "real" malicious code, but recovery often fails (Martignoni *et al* 2007). Therefore, unpacking is a process prior to malware detection. It consists of dumping unpacked image from memory to a file, modifying header and Import Address Table (IAT) and reconstructing PE. Unpacked executable can be achieved statically, dynamically or manually as discussed earlier.

4. Proposed methodology

Our model shown in figure 2 aims to explore randomness in packed executables. The proposed mechanism is based on two approaches *i.e.*, ESCAPE (Entropy SCore Analysis of Packed Executables) (Naval *et al* 2012) and PEAL (Packed Executable AnaLysis) (Laxmi *et al* 2011). ESCAPE utilizes blockwise entropy values and PEAL uses learning algorithm for discriminating packed or native binaries. Figure 2 highlights the steps implemented to achieve our objective. The word 'native' is used here to represent the binaries which are not packed or encoded.

Following are the actions performed to implement proposed approach:

– Malware and benign PE files are considered in packed and native form. The packed dataset is formed using (1) three packers (UPX, ASPACK, PECOMPACT) and (2) XOR-based encoders including shikata ga nai.

---

[3] upx.sourceforge.net/

[4] http://www.aspack.com/

[5] http://www.woodmann.com/crackz/Packers.htm#pecompact

[6] http://www.exploit-db.com/wp-content/themes/exploit/docs/18532.pdf

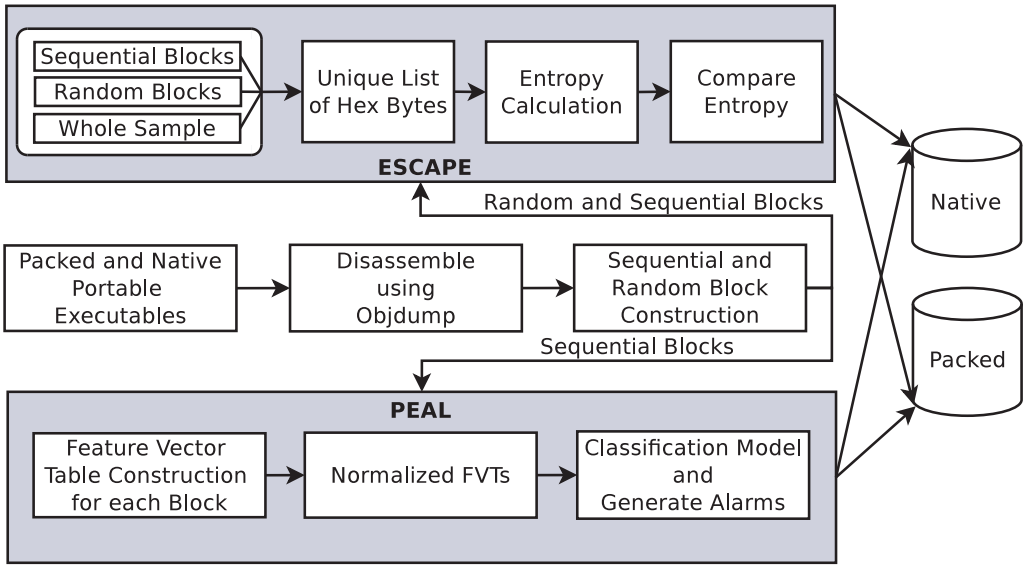


Figure 2. General approach.

- Each dataset is disassembled using Objdump[7] and raw hex byte view is achieved.
- Twenty blocks containing $500, 1000, 1500, \ldots, 10000$ hex bytes are extracted either sequentially or randomly. In sequential mode, we select a contiguous block of 10000 bytes and assign first 500, 1000, 1500, ... 10000 bytes to first, second, third, ... last block respectively. In random mode, contiguous $500 \times k$ bytes are selected for k^{th} block. These blocks are input to both the phases *viz.* ESCAPE and PEAL.
- ESCAPE calculates blockwise (sequential and random mode) and overall entropy of the whole sample. These blockwise entropy values of packed and native samples are compared to distinguish packed and native samples.
- PEAL constructs the feature vector tables (FVT) of unique features extracted from each block. Here, only sequentially extracted blocks are considered to validate our heuristic that "packer code is randomly distributed over portable executable". Bi-classification model is prepared to classify packed and native binaries.

4.1 *System configuration*

The system used for experimentation has an Intel Core i3 processor with 2.40 GHz clock frequency and a 2.8 GiB RAM. GNU gcc 4.3 version compiler, Ether (Dinaburg *et al* 2008) and WEKA[8] are used for implementation.

4.2 *Workload preparation*

Workload is prepared using 970 benign and 474 malware PE files. Only '.exe' files are considered due to their popularity. Malware dataset is constructed by downloading real malware

[7] http://sourceware.org/binutils/docs/binutils/objdump.html
[8] http://www.cs.waikato.ac.nz/ml/weka/

samples from VX Heavens[9] and Offensive Computing.[10] It contains viruses, worms, spywares and trojans. Benign dataset is composed by copying `.exe` samples from `system32` directory of freshly installed Windows XP. Both benign and malware datasets are scanned (Norton Internet Security 2011, Kaspersky and clamAV) to ensure their cleanness and maliciousness respectively.

To form native datasets of benign and malware samples, we used three unpackers: ether, VMUnpacker[11] and GUNpacker.[12] The acquired unpacked samples are further checked using PEiD[13] to reaffirm their nativeness as unpacking techniques may not always be effective (Vilkeliskis 2009; Martignoni *et al* 2007). To form packed workload, each sample has also been packed with (a) UPX, ASPACK and PECOMPACT and (b) XOR-based encoders. Native benign and malware binaries are packed using these packers (UPX, ASPACK, PECOMPACT). In addition to that, we also created one dataset in which the malware samples are encrypted using two encoders *i.e.*, shikata ga nai and one XOR-based encoder implemented in Metasploit". Two levels of encoding are applied on native malware samples to bypass the detection. Metaspoilt has ranked shikata ga nai as excellent encoder because it is highly polymorphic. We utilize this encoder to capture malware samples packed with XOR-based encryption algorithms. We formed nine datasets of which seven are packed/encoded and two are not packed *i.e.*, native. As all of the datasets are referred multiple time in this paper; therefore, each dataset is associated with one label to identify it. Datasets with their labels and nomenclature are as follows.

- Native Benign (N_B)
- Native Malware (N_M)
- Encoded Malware (E_M)
- Packed Benign with UPX (P_B^U)
- Packed Benign with ASPACK (P_B^A)
- Packed Benign with PECOMPACT (P_B^P)
- Packed Malware with UPX (P_M^U)
- Packed Malware with ASPACK (P_M^A)
- Packed Malware with PECOMPACT (P_M^P)

4.3 *Sample disassembly*

The samples are disassembled using `Objdump` that displays information of a binary in raw format such as showing low level assembly code, section-wise disassembled view, hex byte view and headers (file, section and private). Each executable was disassembled to achieve its source code view in raw hex byte format. Blocks are extracted from disassembled code for experimentation. In our experiment, raw hex bytes are considered for experimentation representing the byte-level view of packed and native binaries. Using four consecutive hex bytes as a feature, we

[9] http://vxheaven.org/
[10] http://www.offensivecomputing.net/
[11] http://www.leechermods.com/2010/01/vmunpacker-16-latest-version.html
[12] http://qunpack.ahteam.org/?p=327
[13] http://www.aldeid.com/wiki/PEiD

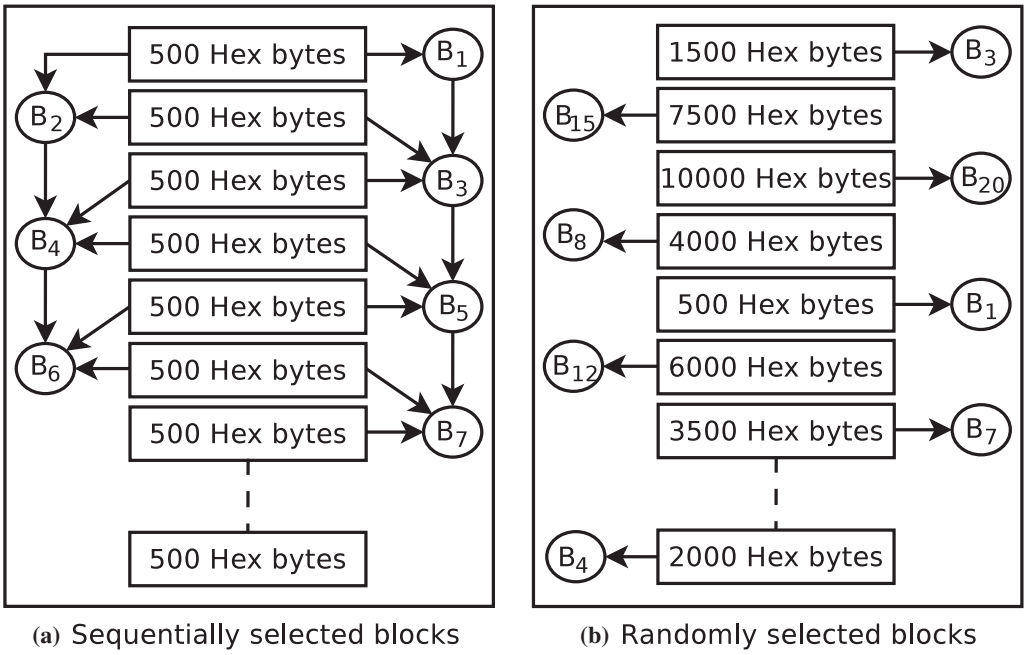
```
Contents of section .text:
 1001000 78960000 f6950000 02960000 1e960000  x...............
 1001010 34960000 48960000 60960000 ec950000  4...H...`.......
 1001020 86960000 98960000 a8960000 ba960000  ................
 1001030 ce960000 e0960000 00000000 11000080  ................
 1001040 00000000 c69b0000 00000000 4e970000  ............N...
 1001050 5a970000 70970000 84970000 a0970000  Z...p...........
 1001060 3a970000 b8970000 c4970000 d0970000  :...............
```

Figure 3. Raw hex bytes.

determine its frequency of occurrence in each sample of the database. Using raw data as features enables us to consider all aspects of binary like opcode mnemonics, registers, data transfer, stack operation and function calls. Figure 3 shows raw hex bytes of `.text` section. The selected region in the figure highlights the features considered for the proposed approach. 0x78960000, 0xF6950000, 0x02960000, 0x1E960000, 0x34960000, . . . are the byte features for which both the models are implemented. A single feature is composed of 4 consecutive bytes. For example, the byte feature 0xF6950000 is made of four consecutive bytes F6, 95, 00, 00. In this paper, we refer this composition of bytes as hexbyte feature.

4.4 *Block construction*

Packed and natives binaries are differentiated using contiguous chunks of hexbytes, each chunk referred to as *block* in this paper. Blocks are considered instead of entire sample to accelerate the proposed approach. A total of 20 blocks are created which contain 500, 1000, 1500, . . . , 10000



(a) Sequentially selected blocks **(b)** Randomly selected blocks

Figure 4. Block construction.

number of bytes and are named as $B_1, B_2, B_3, \ldots, B_{20}$ respectively. Block B_k is composed of $500 \times k$ bytes. This construction helps us analyze a sample blockwise irrespective of the sample size. Blocks are created in two ways to check for randomness (a) sequentially and (b) randomly (refer to figure 4).

4.4a *Sequentially extracted blocks*: In this category, the blocks are selected from the beginning of the file in consecutive manner as shown in figure 4(a). Every k^{th} block is constructed by concatenating next 500 bytes to $(k-1)^{\text{th}}$ block. These blocks are created sequentially one after the other as shown below.

$$B_1 \subset B_2 \subset B_3 \subset \cdots B_k \cdots \subset B_{20}$$

where $B_k = 500 \times k$ bytes and k ranges from 1 to 20.

4.4b *Randomly selected blocks*: Figure 4(b) depicts the construction of blocks randomly. The random blocks are selected in the whole span of PE file. Here also 20 blocks are chosen, each block being a collection of consecutive bytes. The 20^{th} block contains 10000 hex bytes which are positioned sequentially in the file. Random Number Generator Function (RNGF) is used to select the beginning of k^{th} block (consisting of $500 \times k$ bytes) from sample in the range from $[1 \cdots \texttt{number\_of\_lines}]$. This function is synchronized with system clock to provide a different number every time. Randomly selected blocks in a file may or may not be overlapped in a sample.

5. Entropy SCore Analysis of Packed Executables (ESCAPE)

To detect the packed executables using ESCAPE, we utilize entropy *i.e.*, a statistical metric. Entropy is used to measure the randomness in a sample incurred due to packing. Shannon proposed the concept of entropy (Shannon & Weaver 1963). In data analysis, entropy can be used to quantify the randomness of a sequence of bytes (Lyda & Hamrock 2007). Entropy represents the expected information carried by a feature. It offers a way to determine the maximum extent by which a sample is either encrypted or compressed (Goise & Olla 2001). To capture the randomness in a packed sample, the hex byte streams are used in our experiment. Using random generators, packers always insert the code randomly to avoid detection. The archived files are also compressed but the data is highly structured and therefore it is not random (Haahr 1999). The entropy of a discrete random event X is computed using the following formula (Shannon & Weaver 1963):

$$H(X) = -\sum_{i=1}^{n} p(x_i)\ \log_2\ p(x_i) \quad H(X) \geq 0 \text{ and } H(X) \in \mathbb{R}, \tag{1}$$

where $p(x_i)$ is the probability of the i^{th} element x_i of series X consisting of n symbols. The underlying hypothesis of ESCAPE is that "packer code is randomly distributed". The proposed approach is aimed to target the regions affected by packer in a given sample.

Algorithm 1 Blockwise entropy calculation

INITIALISATION:
$B = \{B_1, B_2, \cdots, B_{20}\}$ // Set of *20* blocks
n_i: Number of unique features i^{th} block of a sample.
$F_i = \{F_1, F_2, \cdots, F_{n_i}\}$ // Set of unique features of i^{th} block.
λ: Total number of features in a block.
C_{ij}: Occurrence of j^{th} feature in i^{th} Block.
P_{ij}: Probability of j^{th} feature in i^{th} Block.
E_i: Entropy of i^{th} block.

```
for Every block B_i of sample, where i = 1 to 20 do
    //Constructing Feature Set F_i
    Set C_ij = 0
    Set E_i = 0
    for every j^th feature in B_i, where j = 1 to n_i do
        if F_ij ∈ F_i then
            C_ij = C_ij + 1
        else
            F_i = F_i ∪ F_ij
            C_ij = C_ij + 1
        end if
    end for
    // Block-wise entropy computation
    for j = 1 to n_i do
        //Calculating P_ij
        P_ij = C_ij / λ
        E_i = E_i − P_ij log_2 P_ij
    end for
end for
```

5.1 *Entropy computation*

ESCAPE computes entropy of the entire sample and blocks which are extracted sequentially and randomly. The experiment is done in three steps to check for randomness in the sample under consideration. The blockwise entropy is used to increase probability that at least one (random) region in the sample affected by the packer is captured. Considering blocks reduces data to be handled and improved the entropy computation and execution time.

5.1a *Blockwise entropy calculation*: Algorithm 1 depicts the procedure of computing blockwise entropy. Twenty blocks are extracted as explained earlier. The entropy of each block is computed. A unique list of features corresponding to a block is created and lists the byte features and their frequency of occurrence. The probability of every feature is determined in that block. Finally, the entropy is calculated using Eq. (1). This procedure is followed for all the blocks.

Entropy of dataset. Entropy of k^{th} block in a dataset is determined by averaging the entropies of k^{th} block of all the samples of the dataset. Entropy of k^{th} block of native dataset is compared with respective entropy of packed (UPX, ASPACK, PECOMPACT) as well as encoded dataset.

5.2 *Results*

Entropy values of native and packed samples are tabulated in table 1. Every packed benign and malware dataset is compared with native benign and malware dataset respectively on the basis of blockwise entropy. In table 1, we have listed the entropy scores of B_1–B_6 blocks as similar trends were observed in subsequent higher blocks. Table 1 displays the four categories: (1) Native versus packed benign (sequential mode), (2) Native versus packed malware (sequential mode), (3) Native versus packed benign (random mode) and (4) Native versus packed malware (random mode). The results show that the native samples have low entropy score as compared to packed ones. The highlighted values indicate that from this block onwards the entropy difference between packed and native samples is noticeable. We have selected the threshold value of difference as $\geq$0.5. Also, it can be observed from table 1 that only B_5 or consecutive 2500 bytes are needed for discriminating native from packed in each category. We have also computed the entropy score of the dataset E_M with native malware. We observed that the samples encoded with XOR-based algorithms do not have high entropy regions. So, in this particular case, ESCAPE does not succeed to identify packed and native samples.

5.3 *Inferences*

The following inferences can be drawn from extensive experimentation using ESCAPE:

1. The packed samples have higher entropy values than native samples that is indicative of randomness introduced by packer.
2. Larger variations across blockwise entropy values of a packed sample is suggestive of packer code being randomly distributed over the binary.
3. Any random selection of blocks gives the same results strengthening the notion that distribution of packer code is not region specific.
4. To differentiate a packed and native executable, scanning of entire file is not required as only 2500 bytes are sufficient.

Table 1. Blockwise entropy of packed and native datasets (sequentially and randomly extracted blocks).

Blocks	Dataset	500	1000	1500	2000	2500	3000
Sequentially	N_B	5.77	7.27	8.08	8.43	8.67	9.02
	P_B^U	6.55	7.30	**9.60**	9.29	9.23	9.74
	P_B^A	6.40	7.17	**9.61**	9.54	9.52	10.40
	P_B^P	6.40	7.39	**9.76**	9.65	9.64	10.51
Sequentially	N_M	6.06	7.24	8.81	8.93	8.98	9.79
	P_M^U	6.16	7.16	8.82	9.05	**9.55**	10.57
	P_M^A	6.25	7.24	9.63	9.77	**9.81**	10.67
	P_M^P	6.46	7.27	9.96	9.89	**9.88**	11.04
Randomly	N_B	4.5	5.95	6.15	6.23	7.18	7.38
	P_B^U	4.4	6.98	**7.08**	7.16	8.09	8.8
	P_B^A	5.8	6.11	**7.90**	7.85	8.04	9.24
	P_B^P	5.38	6.72	**7.21**	7.61	8.11	8.84
Randomly	N_M	5.37	6.64	7.61	8.37	8.48	9.09
	P_M^U	6.18	6.98	9.47	9.35	**9.40**	10.33
	P_M^A	6.28	7.16	9.59	9.66	**9.68**	10.5
	P_M^P	6.36	7.13	9.84	9.68	**9.58**	10.55

5. The samples encoded with XOR-based encryption algorithm do not have higher entropy regions. The XOR-based algorithms are entropy-preserving and even after encryption do not increase the entropy of the samples.

6. Packed Executable AnaLysis (PEAL)

PEAL, second approach for discriminating packed from native binaries, is based on feature vector construction through selection process. This approach investigates the diversity of hexbyte features incurred due to packing. Also, we constructed the FVTs in blockwise manner to validate the heuristics of ESCAPE. Initially, every hexbytes of the blocks (B_1, $B_2 \ldots$, B_{20}) of the sample is considered a feature. Through frequency of occurrence across respective blocks of all samples, relevant blockwise features are identified. These features constitute our FVT. As shown in ESCAPE, both sequentially and randomly extracted blocks produce similar results. Therefore, we have considered only sequential mode of blocks' selection. Once features are extracted, we have applied machine learning algorithms on training set to discriminate between packed and native classes of test dataset. In conjunction, we have also applied cross-validation with 2, 3, 5, 7 and 10 folds. Figure 5 shows the work flow of PEAL. For each experiment, following steps are applied.

- For each block of a dataset (packed/encoded/native), feature vector table (FVT) is constructed by selecting block-wise relevant unique hex-byte features.
- Min–Max normalization applied to FVT.
- Normalized FVT used as an input to machine learning classifiers (We have used WEKA).
- Five learning algorithms – *viz.* Naïve Bayes (NB), J-48, IBk, Ada-Boost (AB) and Random Forest (RF) – were applied. Results validated using k-fold cross–validation ($k \in 2, 3, 5, 7, 10$).

In addition to validation, PEAL also compares the applied learning algorithms and k-cross–validation to discriminate among two classes as follows:

1. N_M vs N_B: To determine whether the blockwise hex bytes can differentiate the malware and benign samples.

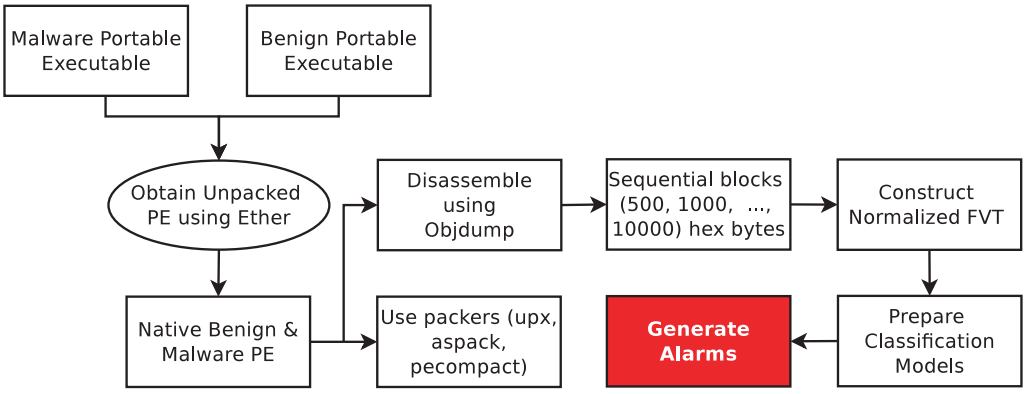


Figure 5. General approach: PEAL.

2. P_B vs N_B: This experiment considers packed and native benign samples. Each of the three packers is applied on native benign samples and then compared.
3. P_M vs N_M: This experiment is carried out between the samples of packed and native malware.
4. P_M vs N_B: This experiment is conducted to check the detection accuracy of our model with packed malware and native benign.
5. E_M vs N_B: This experiment is performed to evaluate PEAL in the presence of encrypted malware. Here, we computed the detection rate of encrypted malware and native benign.
6. E_M vs N_M: This experiment is performed to evaluate PEAL in the presence of encrypted malware. Here, we computed the detection rate of encrypted malware and native malware.
7. P_M vs P_B: Last experiment is done to ensure whether two executables packed with the same packer can be distinguished or not.

6.1 *Classification model*

The classification model is prepared by applying classification algorithms and cross-validation on training and test sets. Implementation results are obtained using WEKA. Sixty percent and forty percent of each dataset are considered as training and test set respectively. Accuracy is determined which represents the correctly classified instances amongst all instances.

6.1a *Classification algorithms*: As discussed earlier, five classification algorithms *namely* Naïve Bayes (NB), J-48, IB*k*, Ada-Boost (AB) and Random Forest (RF) are applied to classify instances from different datasets.

- Naïve Bayes (Sang-Bum *et al* 2006) is a probabilistic algorithm based on Bayes theorem with the assumption that features are independent of each other. This algorithm is used as a learning statistical model for classification. It utilizes all features and analyzes each feature individually.
- J48 is a predictive learning algorithm and prepares a decisions tree. Training data instances are considered for tree formation. Each of the instances is matched with the most similar set of values that can differentiate it clearly.
- IB*k* learning algorithm considers instances to learn and build the model. New instances are identified by searching k neighboring training instances with highest votes.
- Ada-Boost (Freund & Schapire 1997) is an adaptive boosting algorithm. It is a strong classifier composed of a set of weak classifiers to improve their performance. In our experiments, we have used J48 as the base classification algorithm. Each classifier is learned in a step-wise manner and their votes generate the boosted classifier.
- Random-Forest is an ensemble based classifier and that is the reason why it produces better classification results as compared to other classifiers. A random subset is selected from training set with balanced class distribution. The Random Forest classifier accumulates the results from all created trees to produce classification results.

6.1b *Cross-validation*: Cross-validation (Payam *et al* 2009) is a method of evaluating learning algorithms by applying rotations of subsets within a dataset at every fold. k-fold in a cross-validation partitions the data into k subsets and performs analysis on one of the dataset to decrease the variability. Subsequently, k iterations of training and validation are implemented such that for each iteration, a different fold of the data is held-out for validation while the remaining $k-1$ folds are used for learning. The main goal of cross-validation is to

estimate the performance of the learned model from available data using one particular algorithm. Experiments in PEAL have been performed using cross-validation with different folds ($k = 2, 3, 5, 7, 10$). We have observed that 10-fold cross-validation performed better than other folds. Similarly, in (Kohavi 1995) better results were reported with 10-fold cross-validation.

6.2 *Evaluation metrics*

PEAL results are analyzed using evaluation metrics (Vinod *et al* 2012) with four parameters. The parameters are True Positive Rate (TPR), False Positive Rate (FPR), True Negative Rate (TNR) and False Negative Rate (FNR). These parameters are best suitable to evaluate the performance of a detection model. Figure 6 shows a 2×2 contingency matrix to express these four parameters. These metrics are computed using four values *viz.* – TP (True Positives), TN (True Negatives), FP (False Positives) and FN (False Negatives). Here, P and N denote the total number of positive (packed) and negative (native) samples. TP denotes the number of packed samples classified as packed, FP gives the number of native samples classified as packed, FN indicates the number of packed samples classified as native and TN denotes the number of native samples termed as native. Classification accuracy is represented by ACC. For any malware detector, high TPR and TNR values along with low values of FPR and FNR are required. This would ascertain that the scanner is capable of correctly identifying samples as malware or benign. So, the same concept is applicable in our approach that is also addressing a packed malware detection problem.

6.3 *PEAL results*

For each classification experiment, results with PEAL are presented and analyzed in this section. A high TPR and low FPR validates correct classification. Results are shown for B_1 to B_7 only, as no changes in TPR and FPR are observed thereafter. The first experiment is performed to explore the effectiveness of blockwise discrimination of native malware and native benign samples. Table 2 depicts the TPR, FPR and ACC corresponding to each of the considered learning algorithms and with cross-validation folds 2,3,5,7 and 10. Table 2 shows that 10-fold outperforms the other folds of cross-validation and the Random Forest presents better results as compared to other classifiers. The highlighted values in the table depict the accuracy for each

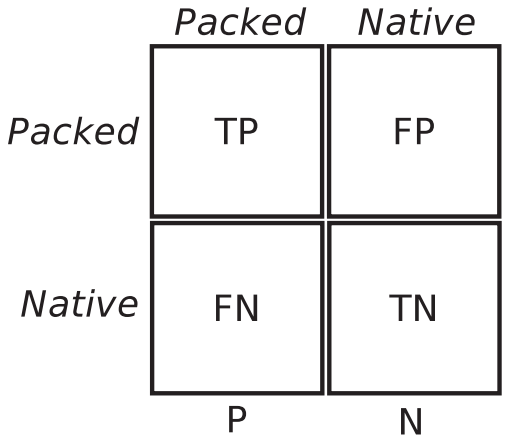


Figure 6. Evaluation metrics.

Table 2. Byte feature classification of N_M vs N_B (Experiment 1).

Bytes	k–fold	RF			AB			IBk			j48			NB		
		TPR	FPR	ACC	TPR	FPR	ACC	TPR	FPR	ACC	TPR	FPR	ACC	TPR	FPR	ACC
B_1 (500)	2	73.8	18.2	77.9	76.7	20.4	78.2	44.9	20.5	62.6	67.2	23.5	72	53.5	18.1	67.9
	3	75.1	18.5	78.3	73.5	22.9	75.4	46.6	16.5	65.4	63.3	19.3	72.2	53.9	18.7	67.9
	5	74.9	16.8	79.1	76.8	20.1	78.4	45.9	17.1	64.7	65.6	20.6	72.6	53.9	19	67.7
	7	74.8	17.1	78.9	73.2	18.1	77.6	47	16.1	65.8	64.4	17.4	73.7	54.1	20.5	67.1
	10	77.2	17.6	79.8	74	16.9	78.6	47.1	16.1	65.9	66	16.7	74.8	55.4	20.2	67.8
B_2 (1000)	2	90.7	15.8	87.5	90.4	15.7	87.4	88.1	17.8	85.2	87.8	17.8	85.1	33.4	10.8	60.7
	3	90.8	13.2	88.8	88.6	11.5	88.5	87.3	16	85.7	89.5	15.2	87.2	33.7	11.9	60.4
	5	92.3	13.1	89.6	90.4	12.8	88.8	89.3	14.7	87.3	87.8	15.2	86.4	33.1	9.9	61
	7	91.6	12.5	89.6	91	13.1	89	88.7	16.2	86.3	87.4	14.6	86.4	33.2	10.6	60.7
	10	92.5	12.8	89.9	90.8	13.1	88.9	90.1	15.3	87.5	89	14.6	87.2	33.3	9.8	61.2
B_3 (1500)	2	90.6	14.5	88.1	89.2	14.6	87.3	90.3	18.2	86.1	87.2	18.2	84.5	68.3	19.5	74.3
	3	91.6	11.9	89.9	89.5	10.6	89.4	92.6	14.8	88.9	88.7	14.7	87	67.1	17	74.9
	5	93.2	11	91.1	91.7	10.7	90.5	93.3	14.4	89.5	90	14.8	87.6	71.2	18	76.5
	7	92.2	10.9	90.7	91.8	10.8	90.5	92.5	15.2	88.7	88.6	13.9	87.4	62.3	16.1	72.9
	10	93.3	11.5	90.9	93.2	11.9	90.7	92.7	14.5	89.1	90	14.5	87.8	60.8	16.3	72
B_4 (2000)	2	92.1	14.1	89	91.2	14.3	88.5	92.5	16.4	88.1	85.7	17.1	84.3	85.7	16.1	84.8
	3	92.4	10.8	90.8	90.9	12	89.5	93.8	14.7	89.6	88	15.3	86.4	85.6	14.6	85.5
	5	93.3	11.2	91.1	91.2	12.7	89.3	94.2	15.9	89.2	88.9	16.1	86.5	85.9	14.6	85.7
	7	93.9	10.8	91.6	92.3	11.3	90.5	94.3	14.3	90.1	88.6	15.1	86.8	85.9	15.2	85.4
	10	94.3	11.2	91.6	93.2	10.9	91.2	94	14.8	89.6	89.6	15.6	87.1	87	15.1	86
B_5 **(2500)**	2	92.1	10.2	91	90.2	12	89.1	92.2	14.6	88.8	90.1	17.6	86.3	84.8	14.5	85.1
	3	91.8	10.1	90.9	91.7	9.4	91.2	93.6	13	90.4	90.1	15.7	87.3	84.6	14.4	85
	5	93.6	9.4	92.1	92.5	8.7	91.9	93.6	13.2	90.3	89.6	14.4	87.6	85	15.8	84.6
	7	92.3	9.5	91.4	93	8.9	92.1	94.2	13.2	90.6	91	15.1	88	85.8	14.7	85.6
	10	93.8	9.7	92.1	92.8	8.8	92	93.7	12	90.9	91.4	14.5	88.5	85.5	15.2	85.2
B_6 (3000)	2	90.8	9.9	90.5	91.9	11.4	90.3	91.6	16.2	87.8	88.8	15.9	86.5	86.5	14.3	86.1
	3	92.8	10.3	91.3	91.1	9.2	91	92.4	13.7	89.4	88.7	14	87.4	85.6	12.9	86.4
	5	93.1	8.9	92.1	92.3	9.7	91.3	93.1	13.1	90.1	90.6	16.4	87.2	85	12.2	86.4
	7	93	9.5	91.8	92.5	9.2	91.7	93.5	13.9	89.8	91	14.3	88.4	85.7	12.4	86.7
	10	93.1	9.6	91.8	92.7	8.6	92.1	93.7	13.2	90.3	91	14.4	88.3	85.3	12.5	86.4
B_7 (3500)	2	92.4	10.8	90.8	91.2	9.1	91.1	92.1	15.1	88.6	89	14.7	87.2	85.8	14.4	85.7
	3	92	8.9	91.6	92.4	9.2	91.6	93.9	14.2	89.9	90.5	13.7	88.4	85.4	13.5	86
	5	92.8	8.5	92.2	92.7	9	91.9	94.4	13.5	90.5	90.1	12.6	88.8	85.3	13.8	85.8
	7	92.9	9.2	91.9	92	8.4	91.8	93.2	13.3	90	89.9	12.9	88.5	85.2	13.1	86.1
	10	94.3	10.2	92.1	92.5	7.5	92.5	94.2	12.2	91.1	89.7	14	87.9	85.3	12.9	86.2

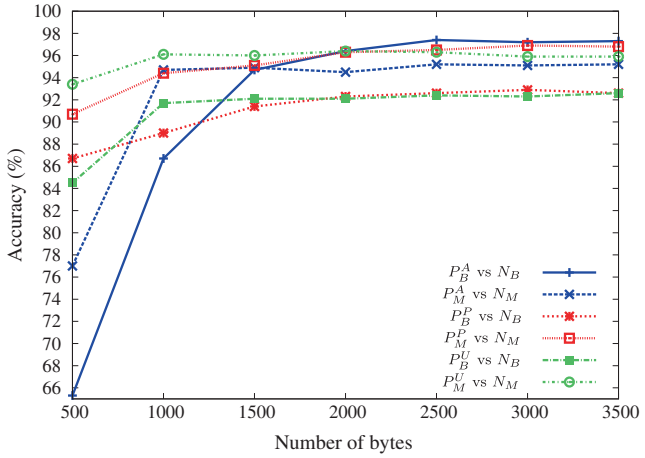


Figure 7. Native vs. packed (Experiment 2 and 3).

listed block. In this case we have achieved a maximum accuracy of 92.1% at B_5 block. Similar trends are observed with higher blocks also.

Figure 7 represents the accuracy obtained from Random Forest classifier with 10-fold cross-validation for each of the FVT constructed for packed and native instances. The blue, red and green lines in figure 7 depict the accuracy corresponding to ASPack, PECompact and UPX packed malware and benign files versus native malware and benign files respectively. It can be seen that the accuracy becomes stable for B_5 onwards. We have achieved accuracy of 97.4%, 95.2%, 92.6%, 96.5%, 92.4% and 96.3% for P_B^A vs N_B, P_M^A vs N_M, P_B^P vs N_B, P_M^P vs N_M, P_B^U vs N_B and P_M^U vs N_M respectively. Experiments 1, 2 and 3 demonstrate that consecutive 2500 bytes are suited for effective discrimination.

The experiment 4 evaluates the performance of our model with packed malware and native benign. This is the most likely scenario in malware detection as malware writers tend to pack

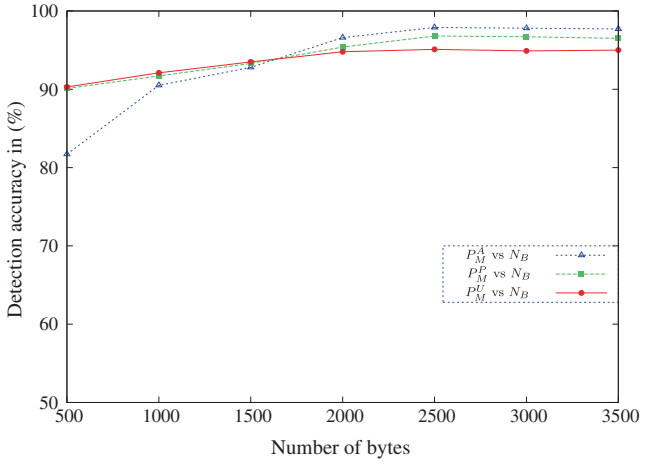


Figure 8. Packed malware vs. native benign (Experiment 4).

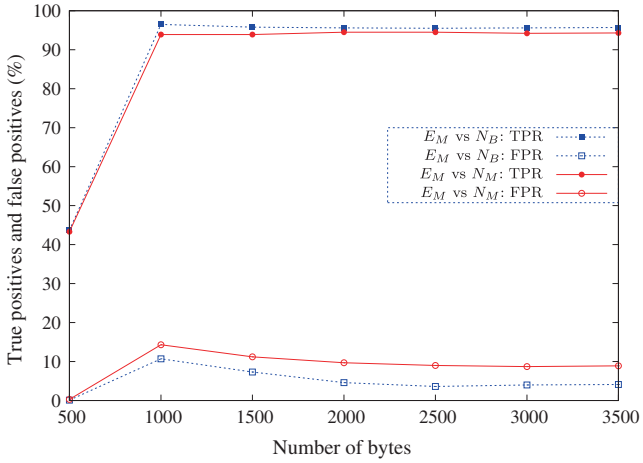


Figure 9. Encrypted malware vs. native (Experiment 5 and 6).

their samples for obfuscation and the chances of applying packing mechanism on benign samples are very low. The software developers apply packing on their benign software for protection against piracy, this number is very small as compared to packed malware. Figure 8 shows that the detection accuracies of 97.9%, 96.8% and 95.1% are achieved for malware samples packed with ASPack, PECompact and UPX respectively. Here, FVT is constructed using block B_5 only.

We conducted experiments 5 and 6 to compute TPR and FPR of our model with encoded malware created using XOR-based entropy-preserving encoders. The XOR encoders encrypt the malicious binary in such a way that the entropy of these samples remains unchanged. Figure 9 shows the TPR and FPR of both the experiments. As it can be observed, PEAL achieves high TPR and low FPR. Although, these encoded malware samples do not show high entropy regions and are able to bypass ESCAPE, encoding creates patterns that are diverse from native sample (either benign or malware).

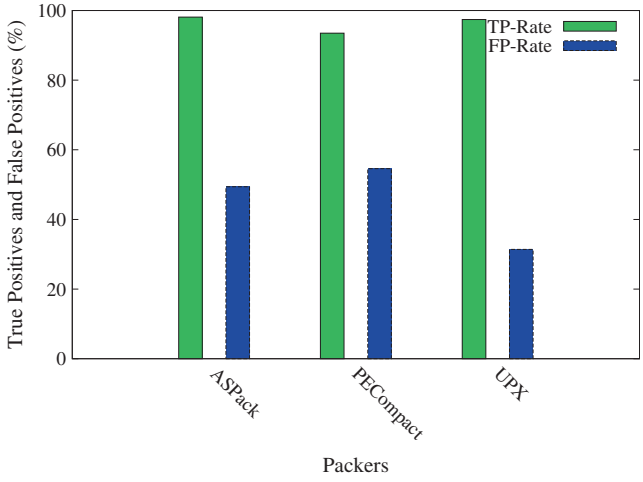


Figure 10. Packed malware vs. packed benign (Experiment 7).

Experiment 7 is conducted to investigate whether the executables packed with the same packer are distinguishable. Experimental results are shown in figure 10 and indicate high TPR and high FPR values even when block B_5 is considered. This indicates the mis-classification. Each packer creates an envelope of its own code around the executables. Two different binaries packed with same packer look similar when disassembled. This confirms the hypothesis that each packer imparts different signature to the sample. Similar observations are noted in case of other blocks.

7. Performance evaluation

In this section, we considered three factors to evaluate the performance of ESCAPE and PEAL. These factors are described as follows.

7.1 *Detection capability*

The detection capability determines the accuracy of our proposed models. The proposed models address the packed malware detection problem that differentiates between packed and native binaries. This detection problem decreases the unpacking overhead of AV-scanners by reducing the number of samples entering into unpacking phase. For ESCAPE, we leverage the fact that the packer code is scattered in the whole span of packed binary. As a result, in packed samples, high entropy values are observed when compared to native samples. But, in case of encoded samples, ESCAPE does not verify our heuristic as these samples are encrypted using entropy-preserving algorithms. On the other hand, PEAL achieves the high TPR and low FPR values in case of packed and encoded samples. The diversity in the hex byte features in packed and encoded samples results into higher detection accuracy with PEAL.

7.2 *System overhead*

The proposed approach is a static approach and works at binary level. The main aim of our approach is to reduce the unpacking overhead of AV scanners by filtering out native samples prior to unpacking phase. Therefore, if we plug our model to AV-scanner, it should not produce excessive overhead in terms of time-complexity. Table 3 shows the running time of our models. ESCAPE calculates the entropy of blocks and PEAL classifies the packed and native samples using FVT constructed blockwise. We have selected the block B_5 as this is the smallest size block yielding high detection accuracy. In table 3, we present processing times in respect to block B_5 containing 2500 hex bytes.

Table 3. Processing time of proposed models.

Average processing time	**ESCAPE (in seconds)**	**PEAL (in seconds)**
Disassembly per sample	0.1465	0.1456
Block extraction time per sample	0.0045	0.0046
Blockwise entropy computation	0.0033	–
FVT construction	–	1.5200
FVT normalization	–	0.0760
Training time	–	1.6700
Total time	**0.1543**	**4.1002**

Table 4. Comparison of PEAL with existing work.

Author/Tool	Features	Detection rate
Perdisci *et al* (2008)	Byte entropy	Not mentioned
Yang-Seo *et al* (2008)	Header	93.6
Lyda & Hamrock (2007)	Byte entropy	Not mentioned
Ugarte-Pedrero *et al* (2011)	Header and heuristics	98.0
PEAL	**Bytes**	**97.4**

Compared to unpacking overhead of AV-scanners, our prototype models show significant improvement in overall performance of AV-scanners. The average total time to compute the blockwise entropy of a sample is ∼0.15 s which is negligible. The blockwise entropy of packed sample is higher than the native samples. But, malware packed with XOR-based encoder does not contain higher entropy regions therefore ESCAPE can only perform better with the samples packed with popular packers *i.e.*, UPX, ASPack and PECompact. On the other hand, PEAL can discriminate packed or encoded malware samples from native samples with higher detection accuracy. The overall processing time of PEAL is ∼4.1 s. This time is inclusive of training time which is one time cost and will not affect the testing phase of PEAL. ESCAPE is more time-efficient than PEAL but does not succeed with samples encoded with XOR-based encoders. In order to provide a generic solution, a hybrid approach can be used, ESCAPE for samples packed with popular packers *i.e.*, UPX, ASPack and PECompact and PEAL for encoded malware samples.

7.3 *Comparison*

Table 4 compares PEAL with other packer detection approaches. It is clearly observed that the performance of PEAL is considerable to other approaches. This comparison of PEAL also proves that blockwise discrimination of packed binaries yields into the detection accuracy of 97.4%.

8. Related work

In (Lyda & Hamrock 2007), authors have proposed a tool named as `bintropy` which discriminates the packed file from encrypted ones by noticing statistical variance in byte streams using entropy. They have used fixed size blocks of 256 bytes to reduce the time overhead. The authors also mentioned limitations of their proposed approach as it gives false negatives and false positives while processing executables of size more than 500kbytes. In (Han *et al* 2009), authors have proposed an entropy based approach which classifies the packed and native files. They have claimed that samples having entropy higher than 6.85 must be categorized as packed. They have compared their tool REMINDer with PEiD and MRC and reported higher detection rate of 97.5%. Authors in (Yang-Seo *et al* 2008) have described a mechanism "PHAD" to detect packed samples by analyzing PE Headers. Their approach differentiates packed and native samples by computing Euclidean Distance of `characteristic vector` of PE header. Authors have extracted the characteristics employed by packers. In (Ugarte-Pedrero *et al* 2011), authors have proposed anomaly detection approach based on structural features to classify packed and native executables. Authors have performed three distance measuring methods viz. Euclidean,

Manhattan and cosine and reported 99% of detection rate. They have validated their approach using 5-fold cross-validation. Authors in (Shafiq *et al* 2009) have presented PE-MIner that is a non-signature based method to identify packed and native PE files. They have used a classification model which consists of feature selection methods (RFR, PCA, HWT) and classification methods (IBk, NB, J48, SMO, Ripper). PE header features are considered in PE-Miner as prominent features for identifying packed and native executables with 99% accuracy and 0.5% false alarm rate. In (Perdisci *et al* 2008), authors have proposed MCBoost that differentiate native and packed samples by considering n-gram of code section, PE file structural features and n-gram of a whole Windows PE file. The authors have combined multiple classifiers to improve the accuracy of MCBoost. They have claimed classification accuracy of 87.3% and an AUC equal to 0.977.

Numerous tools have been also designed to identify packed executables for malware analysis. PEiD (Packed Executable IDentifier) is one example of such tools. PEiD is a signature-based method available with GUI support to identify packers. exeInfo[14] is a win32 PE identifier for detecting packers and saves overlay as external file. Detect it Easy (DiE) offers PE file analyses features like import functions, determining PE section, detects packer and so on. It also uses the signature-based detection method like PEiD to detect a packer. The packer detection technique of DiE is an intellectual property. The ProtectionID tool is used to provide the CD lock information. Besides, it also supports many features including packer detection. RDG packer detector is a tool that offers entropy and file checksum calculation with features of packer detection. This tool has been developed by RDGMax and the signature can also be generated manually.

9. Conclusions

Packing has been proved a boon for malware authors as it saves their time and effort for creating new malware. Packed malware needs to be unpacked for further analysis and reverse engineering. Available unpacking approaches provide solution at the expense of computation speed. Therefore, the malware detectors require a technique which overcomes the execution overhead caused by unpacking engines. In this paper, we have developed a fast and efficient approach which can be plugged in with existing AV scanners to improve their performance. Our proposed models reduce the total number of samples entering into the unpacking phase to enhance the performance of AV scanners. We observed that packed code is having higher entropy than the native binary code but the samples encoded with XOR-based algorithms do not yield into higher entropy. Therefore, to develop a generic solution and to validate the inferences of ESCAPE, we proposed PEAL model. To detect these packed binaries, entire sample is not needed only a fraction of hex bytes (a block containing 2500 or more bytes) is sufficient. These blocks can be extracted either sequentially or randomly from the sample. We obtained results which indicate that block B_5 can be selected as a benchmark to discriminate packed and native samples. The processing time of our model is negligible in comparison to the time consumed by unpacking engine for processing a sample. The experimental results indicate that to classify sample into packed and native classes blockwise hex bytes can be used. The proposed approach is evaluated with three different packers and two XOR-based encoders and achieves comparative detection accuracy.

---

[14] http://exeinfo.atwebpages.com/

References

Avgerinos T, Rebert A, Cha S K and Brumley D 2014 Enhancing symbolic execution with veritesting. In: *Proceedings of the 36th International Conference on Software Engineering* ACM, Hyderabad, India, pp. 1083–1094

Brosch T and Morgenstern M 2006 Runtime packers: The hidden problem. In: *Proceedings of Black Hat USA, Black Hat*, www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf

Bohne L 2008 Pandora's Bochs: Automatic unpacking of malware. In: *PHDTHESIS*, pp. 1–121

Christopher A B 2010 *Maitland: Analysis of packed and encrypted malware via paravirtualization extensions* in https://dspace.library.uvic.ca/handle/1828/3866, pp. 1–82

Coogan K, Debray S, Kaochar T and Townsend G 2009 Automatic static unpacking of malware binaries. In: *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE '09)*, IEEE, pp. 167–176

Dinaburg A, Royal P, Sharif M and Lee W 2008 Ether malware analysis via hardware virtualization extensions. In: *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, ACM, pp. 51–62

Freund Y and Schapire R E 1997 A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.* 55:1, Academic Press Inc.; Orlando FL, USA; pp. 119–139

Goise F and Olla S 2001 Entropy methods for the Boltzmann equation. In: *lectures from a special semester at the Centre Émile Borel Institute* Springer, Poincarè, Paris, pp. 1–14

Haahr M 1999 *An introduction to randomness and random numbers* in www.random.org/essay.html Random.org

Han S, Lee K and Lee S 2009 Packed PE file detection for malware forensics. In: *Proceedings of 2nd International Conference on Computer Science and its Applications (CSA'09)*, IEEE, Jeju Island, Korea, pp. 1–7

Jacob G, Comparetti P M, Neugschwandtner M, Kruegel C and Vigna G 2013 A static, packer–agnostic filter to detect similar malware samples. In: *Proceedings of International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Springer Berlin Heidelberg, pp. 102–122

Jeong G, Choo E, Lee J, Bat-Erdene M and Lee H 2010 Generic unpacking using entropy analysis. In: *Proceedings of 5th International Conference on Malicious and Unwanted Software (MALWARE '10)* IEEE, pp. 98–105

Kang M G, Poosankam P and Yin H 2007 Renovo: A hidden code extractor for packed executables. In: *Proceedings of the 2007 ACM Workshop on Recurring Malcode (WORM '07)* ACM, New York, USA, pp. 46–53

Kevin T 2010 *Malware validation techniques.* In: http://blogs.cisco.com/security/malware_validation_techniques/

Kohavi R 1995 A study of cross-validation and bootstrap for accuracy estimation and model selection. In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95)* 2, Morgan Kaufmann Publishers, San Francisco, CA, USA, pp. 1137–1143

Laxmi V, Gaur M S, Faruki P and Naval S 2011 PEAL-packed executable analysis. In: *Proceedings of the 2011 International Conference on Advanced Computing, Networking and Security (ADCONS'11)* Springer, pp. 237–243

Lyda R and Hamrock J 2007 Using entropy analysis to find encrypted and packed malware. *IEEE Security and Privacy* 5:2, Piscataway, NJ, USA, pp. 40–45

Martignoni L, Christodorescu M and Jha S 2007 OmniUnpack: Fast, generic, and safe unpacking of malware. In: *Proceedings of Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)* IEEE, pp. 431–441

Naval S, Laxmi V, Gaur M S and Vinod P 2012 ESCAPE: Entropy score analysis of packed executable. In: *Proceedings of the Fifth International Conference on Security of Information and Networks (SIN'12)* ACM, New York USA, pp. 197–200

Payam R, Lei T and Huan L 2009 Cross-validation. In: *Encyclopedia of Database systems* pp. 532–538

Perdisci R, Lanzi A and Wenke L 2008 McBoost: Boosting scalability in malware collection and analysis using statistical classification of executables. In: *Proceedings of Computer Security Applications Conference (ACSAC 2008)* pp. 301–310

Royal P, Halpin M, Dagon D, Edmonds R and Lee W 2006 PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In: *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC '06)* IEEE, Washington, DC, USA, pp. 289–300

Sang-Bum K, Kyoung-Soo H, Hae-Chang R and Sung-Hyon M 2006 Some effective techniques for naive bayes text classification. *IEEE Transaction on Knowledge and Data Engineering* 18:11, pp. 1457–1466

Saxena P, Poosankam P, McCamant S and Song D 2009 Loop-extended symbolic execution on binary programs. In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* ACM, Chicago, IL, USA, pp. 225–236

Shafiq M Z, Tabish S M, Mirza F and Farooq M 2009 PE-Miner: Mining structural information to detect malicious executables in realtime. In: *Recent advances in intrusion detection* 5758, Springer, pp. 121–141

Shannon C E and Weaver W 1963 *The mathematical theory of communication*. University of Illinois Press

Ugarte-Pedrero X, Santos I and Garcia-Bringas P 2011 Structural feature based anomaly detection for packed executable identification. In: *Proceedings of Computational Intelligence in Security for Information Systems (CISIS'11)*, LNCS, pp. 230–237

Vilkeliskis T 2009 *Automated unpacking of executables using dynamic binary instrumentation*. http://vilkeliskis.com/_static/research/2009-unpackdbi-paper.pdf pp. 1–14

Vinod P, Laxmi V and Gaur M S 2012 REFORM: Relevant feature for malware analysis. In: *Proceedings of Sixth IEEE international conference of security and Multimodality in Pervasive Environment (SMPE-2012)* Fukuoka, Japan, pp. 26–29

Wei Y, Zheng Z and Ansari N 2008 Revealing packed malware. In: *IEEE Security and Privacy* 6:5, pp. 65–69

Yang-Seo C, Ik-kyun K, Jin-Tae O and Jae-Cheol R 2008 PE File Header analysis-based packed PE file detection technique (PHAD). In: *Proceedings of International Symposium on Computer Science and its Applications (CSA '08)* pp. 28–31