A FAST RANDOMNESS TEST THAT PRESERVES LOCAL DETAIL

Tim Ebringer[1]
The University of Melbourne, Australia

Li Sun, Serdar Boztas
RMIT University, Australia

Email tim.ebringer@gmail.com, li.sun@ca.com, serdar.boztas@ems.rmit.edu.au

ABSTRACT

Randomness tests, sometimes called 'entropy' tests, can be a fast way to estimate whether a file is packed. However, most algorithms we have seen simply give an overall randomness value for the entire file. We present an algorithm that is not only fast, but preserves local detail, so a plot can be made of an entire file showing areas of high randomness and low randomness. Areas of low randomness (code and header, in the case of a PE file) show as lower points on the plot, whereas areas of high randomness (encrypted or compressed data) show as high areas.

We present local detail-preserving entropy plots for a variety of packers, showing that many appear to pack files in a distinctive manner. This seems to be because compressed data and the code that unpacks it tends to be placed in the same relative location in the packed file, leading to a kind of signature based on a 'randomness signal'. We give algorithms which have shown early promise in comparing the entropy signals of various packed files.

Finally, we are able to visualize the work of a packed program as it unpacks itself by placing breakpoints on decompression/decryption loops, dumping memory and performing our detail-preserving randomness analysis on the dump.

INTRODUCTION

Modern malware is usually wrapped in layers of compression and 'encryption'[2], often more than once. Interspersed with the encryption is a smattering of anti-debug, anti-dump, anti-emulation and anti-trace techniques, intended to make the malware more difficult to reverse engineer. Somewhat unfortunately, whilst it is technically difficult for an ordinary software developer to add such gruesome traits to their application, a plethora of tools have emerged to perform this function as a post-processing step. The ones that cause the anti-malware community particular misery are those that are sold commercially as an anti-piracy measure.

A commonly used metric to quickly ascertain whether a file has been packed is to examine the byte frequency distribution of a sample file. The byte frequency distribution in executable files is highly skewed, however, heavy use of compression or encryption intuitively should bring the bytes closer to a uniform distribution. Figure 1 shows a sorted byte distribution for the venerable *Windows* 'calc.exe', for the same program after being packed with UPX 2.03w, and for random bytes. The inference is the flatter the slope, the more likely the executable is packed.

The term 'entropy' is commonly used in the anti-malware community to broadly refer to how far a sample set of bytes deviates in frequency from a purely random distribution. Different scientific disciplines have different definitions of entropy. In thermodynamics, for example, entropy is a measure of the unavailability of a system's energy to do work. However, the formal definition of entropy more commonly used in cryptography, communications theory and coding is due to Claude Shannon, and is often called *Shannon Entropy* to distinguish it from other definitions.

Shannon Entropy is an information-theoretic measure of the amount of uncertainty in a variable [1]. It was introduced in Shannon's seminal works on communication [2, 3], and has found widespread use. The formal definition is as follows:

Let the random variable X take values from some set $\{x_1, \dots, x_n\}$. The value x_i occurs with probability $p(X = x_i)$, where $\sum_{i=1}^{n} p(X = x_i) = 1$

The *entropy*, or *uncertainty* of x is

$$H(X) = -\sum_{i=1}^{n} p(X = x_i) \log_2 (p(X = x_i))$$

Whilst this remains the formal definition of entropy, the heuristic commonly desired from the anti-malware community is a convenient measure of 'how random are these data', with the expectation that compressed or encrypted data will exhibit a higher degree of randomness than code or text.

In the research community, a large body of work has been devoted to answering this question in the context of random number generators, in order to assess how 'good' they are. Knuth devotes a significant section of *The Art of Computer Programming* to randomness tests [4], and the DIEHARD [5] and DIEHARDER [6] battery of randomness tests have enjoyed

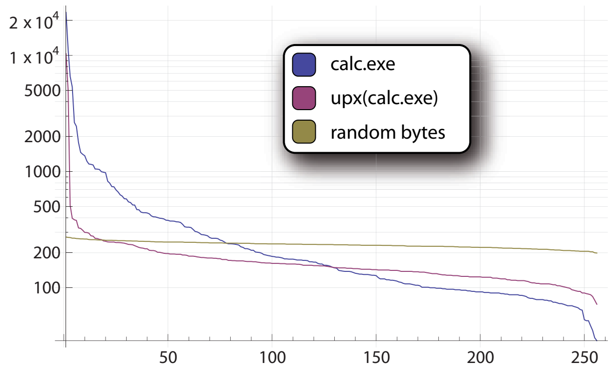

Figure 1: Byte frequency distribution of calc.exe, UPX 2.03w packed calc.exe and random bytes.

[1] This work was performed whilst Ebringer was an employee of *CA*, working in the *CA Labs* research division. Sun is presently a Ph.D. candidate sponsored by *CA Labs*.

[2] The 'encryption' performed is often very weak, sometimes simply an XOR with a fixed byte. Since the key must always exist for the decrypted malware to function, it could be regarded as a simple encoding, however, it does practically add a layer of obfuscation.

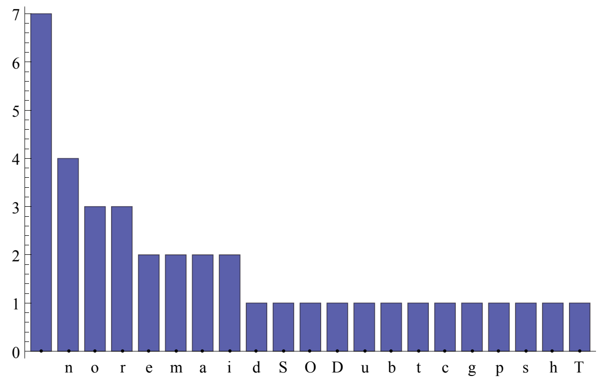

Figure 2: The frequency histogram of 'This program cannot be run in DOS mode'.

considerable popularity over the years as an effective suite of tools for measuring how truly 'random' given binary data is.

When looking for a heuristic, however, these tests have the dual disadvantage that (a) they require significant amounts of data (at least 128 MB in the case of DIEHARD); and (b) are not particularly useful at distinguishing highly random parts of the file from less random parts. For example, they will indicate, correctly from the point of view of the entropy definition, that even a heavily packed PE file is not particularly 'random', due to such non-random features as the PE header, and fields of zeroes that nest in between PE structures.

This work illustrates that by adjusting some simple and well-known algorithms and data structures from the compression community, useful heuristic measures of the amount of 'randomness' in different parts of a sample executable program can be derived. We present results that show that there is utility in using these plots for classification purposes. Finally, we show that with the use of a programmatically controlled debugger and a process memory dumper, we can visualize the operation of a runtime packer as it unpacks itself.

Huffman coding

Huffman coding [7] is a very well known and easy to implement lossless data compression algorithm. It works by constructing an extended binary tree based upon the frequency of symbols. For example, for the following unreasonably famous string:

'*This program cannot be run in DOS mode*'

The frequency table for the symbols is shown in Figure 2, with the space character the most represented, having seven occurrences.

A tree is constructed by starting only with leaves having the weight of their respective frequency. A parent node is introduced for two lowest weight nodes, the internal parent node having as its weight the sum of its two children. So, 't' and 'D' are made the children of a new internal node of weight 2. Only the new node is considered for re-merging. An example tree is shown in Figure 3.

The Huffman encoding for a character is simply a traversal of the tree to the leaf-character, where a '0' represents a left branch, and '1' a right branch, so the binary encoding for 't' is 101110, but a common character, such as a space, is only 111. In this case, the encoding for a space character is three bits, whilst a 't' character requires six.

ALGORITHMS

Two versions of the randomness scanning algorithm are presented. The first generates a fixed number of sample points, which is useful for comparative and storage purposes, but can mask detail on large files, and over-represent detail on smaller files. The second is a 'sliding-window' approach which does not dilate or contract features, at the expense of an output length which is dependent on the file, and thus less convenient for comparative purposes.

Both algorithms first require that the Huffman tree be constructed for the entire file, using bytes as symbols. Operating on bytes as symbols appears the most intuitive approach as it causes the tree to remain small, and the byte is the fundamental building block of most executable programs. There are

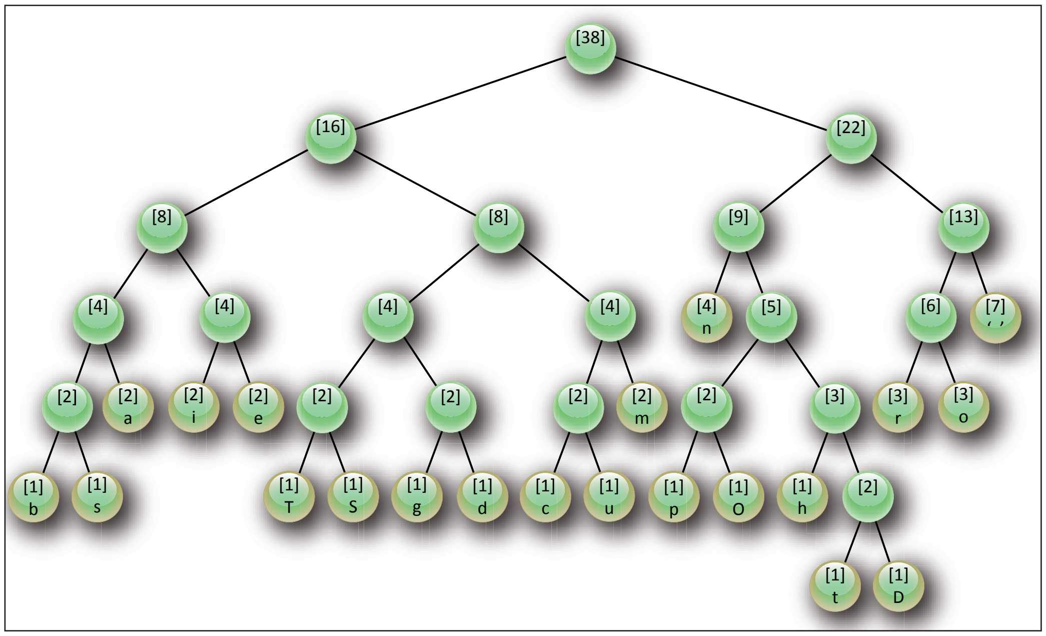

Figure 3: Huffman tree of 'This program cannot be run in DOS mode'.

well-known algorithms to build the Huffman tree, including ones that run in linear time. The time required for file I/O should completely dominate the time required for construction of the Huffman tree.

Algorithm 1: fixed sample count

```
Generate fixed number of randomness measurements for a file

INPUT: An array of bytes $b_1, \dots, b_n$ and a sample count $s$.
OUTPUT: An array of $s$ samples of the randomness, ranging from 0.0 to 1.0.
1. Construct the Huffman tree for the input bytes.
2. Construct an array $e_1, \dots, e_{256}$ containing the encoding length for each of the input bytes.
3. Compute sample offsets $o_1, \dots, o_{s+2}$ as the following:
   a) $o_1 \leftarrow 1$
   b) $o_{s+2} \leftarrow n$
   c) For $i$ from 1 to $s$, do the following:
      i) $o_{i+1} \leftarrow \text{round}\left(\frac{ni}{s+1}\right)$
4. For $i$ from 1 to $s$, do the following:
   a) $r_i \leftarrow \sum_{j=o_i}^{o_{i+2}} e_j$
5. Rescale $r_i$ between 0.0 and 1.0, where $\min(r_i)=0.0$ and $\max(r_i)=1.0$.
```

For the case of a finite alphabet (2^8 in our case) and rational-valued probabilities, it is possible to construct a Huffman tree in linear time, by using e.g. bucketsort [8]. All other steps are also linear in the length of the file, hence the whole algorithm is linear. The advantage of this algorithm is that it outputs a fixed set of samples for different length files, thereby enabling easy comparison. The visual impact is that similar features on different sized files appear as if they were dilated or contracted when plotted together.

Algorithm 2: sliding window

```
Generate randomness measurements for a file, output proportional to file length

INPUT: An array of bytes $b_1, \dots, b_n$, a window size $w$ and a skip size $a$.
OUTPUT: An array of $\left\lceil \frac{n-w}{a} \right\rceil$ samples of the randomness, ranging from 0.0 to1.0.
1. Construct the Huffman tree for the input bytes.
2. Construct an array $e_1, \dots, e_{256}$ containing the encoding length for each of the input bytes.
3. For $i$ from 1 to $\left\lceil \frac{n-w}{a} \right\rceil$, do the following:
   a) $r_i \leftarrow \sum_{j=a(i-1)+1}^{a(i-1)+w+1} e_j$
4. Rescale $r_i$ between 0.0 and 1.0, where $\min(r_i) = 0.0$ and $\max(r_i) = 1.0$.
```

This version of the algorithm produces an output length dependent on the input file size. The advantage of this approach is that it can be used to look for features of a specific size, for example cryptographic keys, by setting the window size w to the size of the interesting feature. However, it is less useful for comparative purposes, and can produce a lot of data.

Pruning

Pruning selects a partial output set from the sliding window algorithm so that a vector of the same size can be extracted. This simplifies comparison between input samples of different length. Unpacking code generally has less randomness than the encrypted or compressed data, so pruning is intended to retain data of low randomness and eliminate data of high randomness. Four types of pruning are proposed here. They are called ‘First’, ‘Smallest’, ‘Ordered smallest’ and ‘Trunk’. For a list of i values, the First pruning retrieves the first N values from the output. Both the Smallest and Ordered smallest pruning methods sort the output first. While the smallest pruning only gets the N smallest values, the Ordered smallest pruning not only gets the N smallest values but also lists them in their original order. The

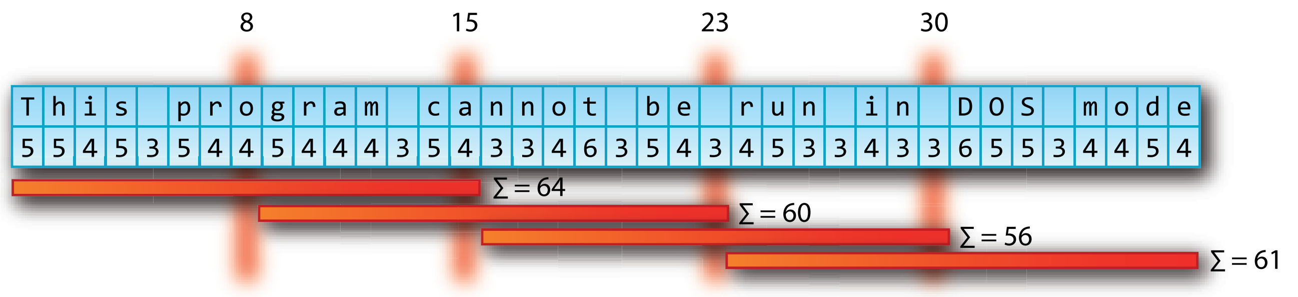

Figure 4: Illustration of fixed sample point algorithm, where sample count is 4.

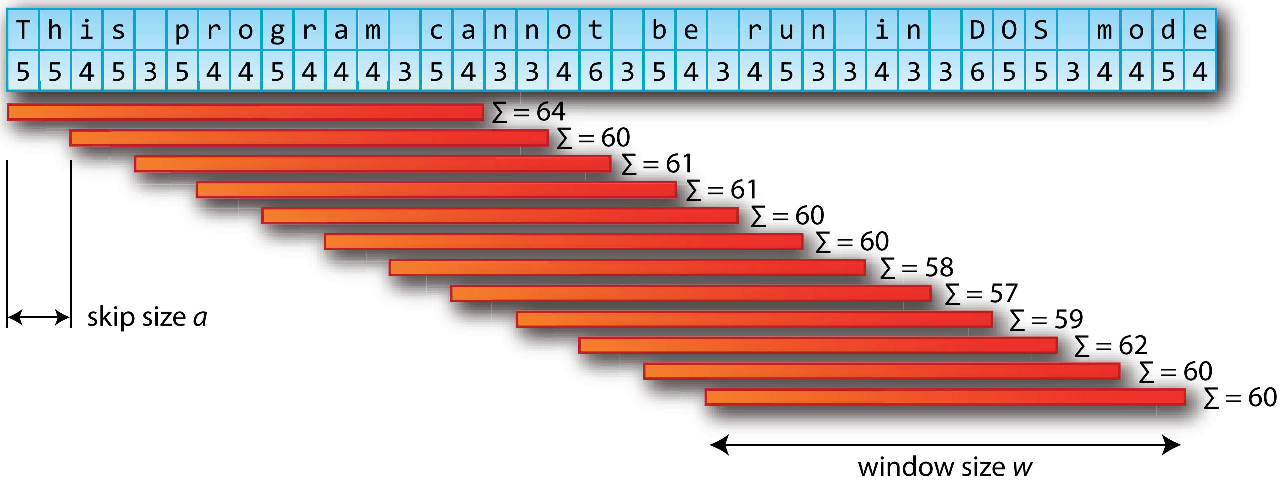

Figure 5: Illustration of sliding window algorithm where window size is 15 and skip size is 2.

Trunk pruning removes the middle part of the output. In other words, it keeps $N/2$ values from the beginning and $N/2$ values from the end of the output. As an example, using I_i, O_f, O_s, O_o, O_t to represent the input, output of the First, Smallest, Ordered smallest and Trunk pruning heuristics respectively:

For $I_{10} = \{1, 3, 4, 9, 8, 10, 6, 7, 2, 5\}$, If $N = 6$:

$O_f = \{1, 3, 4, 9, 8, 10\}$;

$O_s = \{1, 2, 3, 4, 5, 6\}$;

$O_o = \{1, 3, 4, 6, 2, 5\}$;

$O_t = \{1, 3, 4, 7, 2, 5\}$.

EXPERIMENTS AND RESULTS

Randomness scanning

Experiments have been carried out intensively on six packers with the whole collection of UnxUtils binaries [9]. These were selected because they are utilities of reasonable complexity, and since they are command-line-based, they do not block on user input. This is particularly useful when automating debugging. Furthermore, the collection contains a good diversity of large, medium and small programs. The collection has 116 executable files whose file size range from 3 KB to 191 KB. The six packers

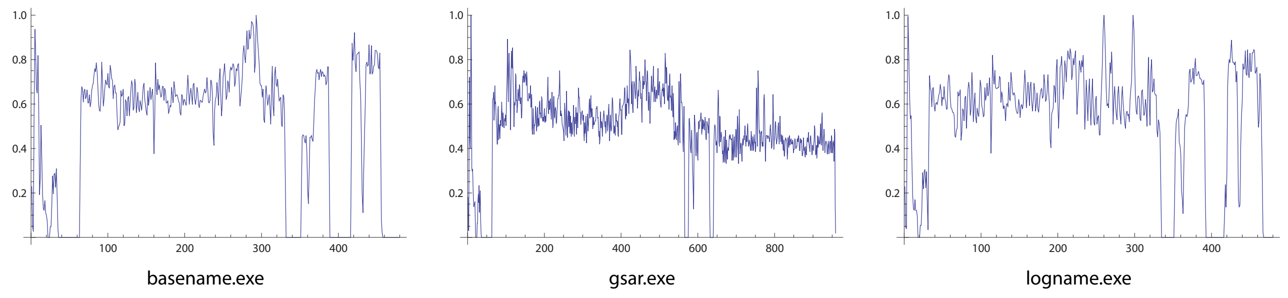

Figure 6: Sliding window randomness scan for clean files.

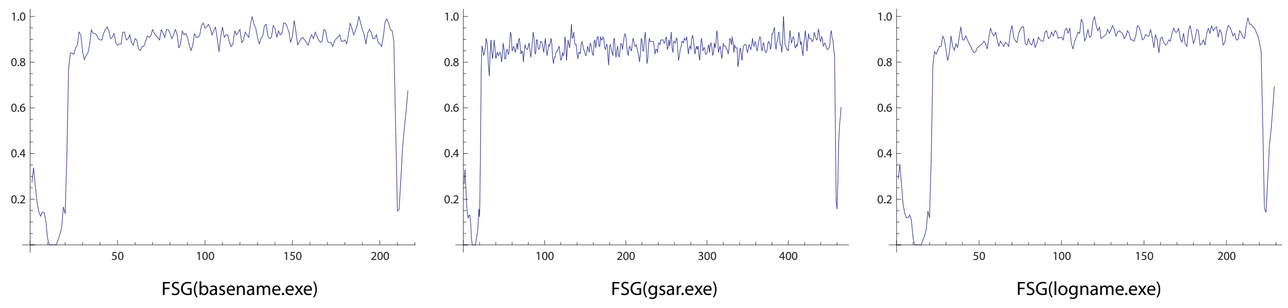

Figure 7: Sliding window randomness scan for FSG 2.0 packed files.

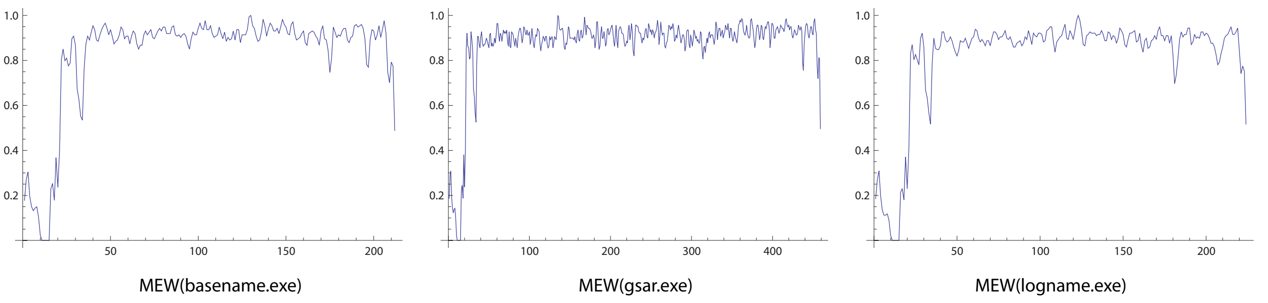

Figure 8: Sliding window randomness scan for MEW 1.1 packed files.

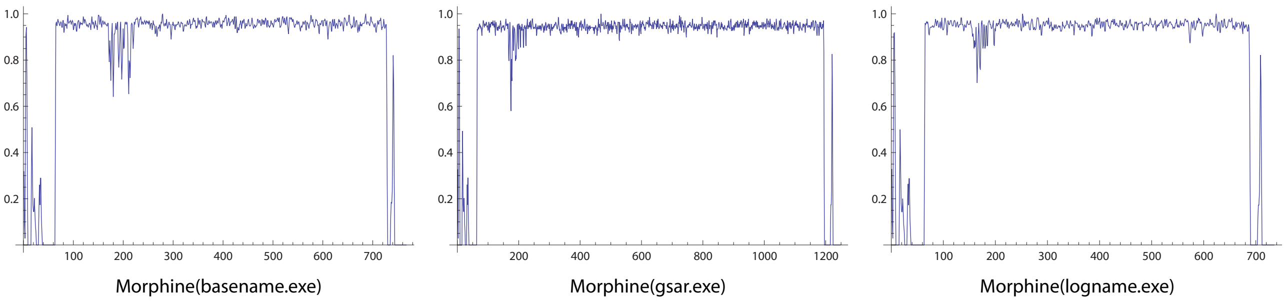

Figure 9: Sliding window randomness scan for Morphine 2.7 packed files.

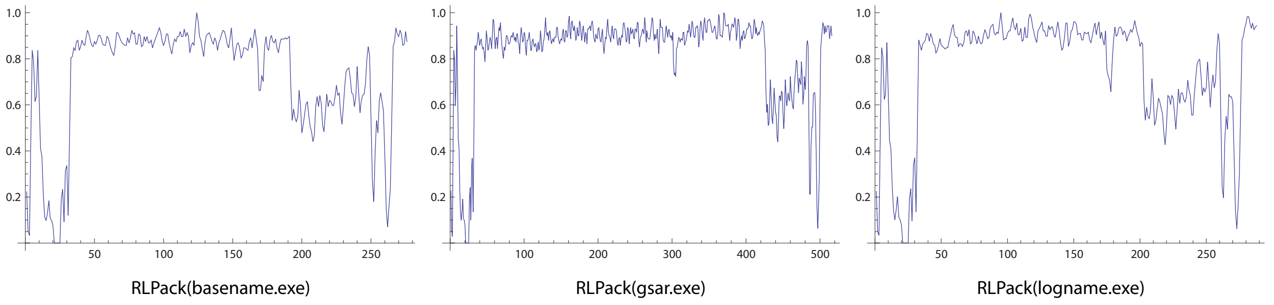

Figure 10: Sliding window randomness scan for RLPack 1.19 packed files.

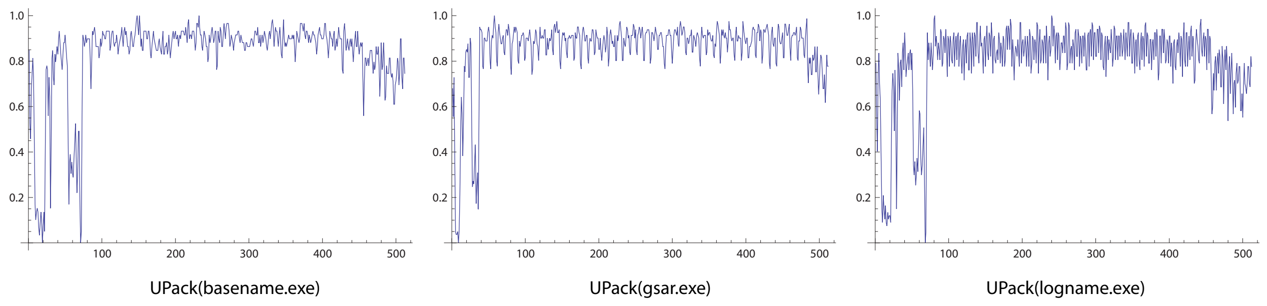

Figure 11: Sliding window randomness scan for UPack 0.399 packed files.

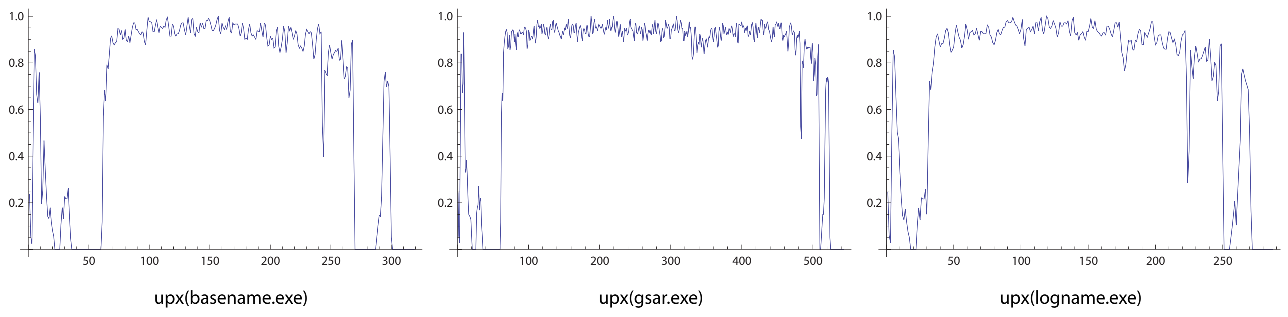

Figure 12: Sliding window randomness scan for UPX 2.03w packed files.

used are FSG 2.0, Mew 11, Morphine 2.7, RLPack 1.19, Upack 0.399 and UPX 2.03w.

Three steps are involved in the 'randomness scanning' experiment. Firstly, for each packer, each file in the UnxUtils set is packed with this packer. Then for each packed file, randomness values in different parts of the file are measured using both the fixed sample count algorithm and the sliding window algorithm. In the fixed sample count algorithm, the sample count *s* is set to 512 to balance the effect of over-representing detail of the small file and under-representing detail of the big file. In the sliding window algorithm, the window size *w* is set to 32 bytes (256 bits) and the skip size *a* is 16, so samples overlap. The final step is to plot the output.

Figures 6–12 show plots of three clean files 'basename.exe', 'gsar.exe' and 'logname.exe' and their corresponding packed files using the sliding window algorithm.

Figures 6–12 preserve local detail by showing areas of relatively high and low randomness across the file. Where a plot bottoms out is usually a region filled with zero (0x00) bytes. High, flat sections of the plot tend to indicate highly random data, often indicating compressed or encrypted data. Note the similar plots for the same packer, even when compared by eye; it is clear that by using this technique, some packers seem to produce a distinctive plot. This suggests a kind of signal which may be used for packer classification.

Packer classification

The randomness test technology can be applied on a packer classification (PC) system. There are two essential steps in a PC system: characteristic extraction and identification. Characteristic extraction retrieves the 'characteristic' of the file which represents the file in a distinctive way. This permits the file to be compared with the signal of candidate packers.

As shown above, the randomness scanning of a file can provide a certain kind of 'characteristic' signal. Using this strategy, a file is represented as an *n*-dimensional vector of randomness values $e=\{r_{e1}, r_{e2}, \dots, r_{en}\}$ where r_{ei} is the randomness value at position *i* in the file. Similarly, a packer's signature is also represented as an *n*-dimensional vector $S=\{r_{s1}, r_{s2}, \dots, r_{sn}\}$ where r_{si} is the randomness value at position *i* in the signature. The value of r_{si} can be generated using the average value of a set of training data which are packed with the same packer. Therefore, for a set of *N* files $E = \{e_1, e_2, \dots, e_N\}$, where e_j is a single file, each value in the signature is computed as:

$$r_{si} = \sum_{j=0}^{N} r_{e_j i} / N$$

Identification refers to the way the packed file is examined and the packer applied to it is identified. The main step is measuring the distance between the sample file and a packer's signal. If the measure is effective, the correct packer will have the smallest distance. For example, if the distances between the file and the signature of packer A, packer B and packer C are 2, 1 and 3 respectively, then the file is regarded as being packed with packer B. For an unknown packer, the lowest distance value will be above a certain threshold.

A wide range of distance measures have been suggested and experimented with. This paper evaluates two commonly used measures. They are sum-of-squares distance (SSD) and the Cosine distance measures.

In both measures, an 'entropy' (randomness) file is represented as a vector *e* and a packer's signature is also represented as *S* as described above. While the SSD measures the distance between each point of the file vector *e* and the signature vector *S*, the Cosine distance measures the angle between two vectors. The smaller this value, the closer are two vectors. They are represented as follows respectively:

$$SSD\ (e,S) = \sqrt{(r_{e1} - r_{s1})^2 + (r_{e2} - r_{s2})^2 + \dots + (r_{en} - r_{sn})^2}$$

and

$$Cosine\ (e,S) = \cos^{-1}\frac{e \cdot S}{|e||S|} = \cos^{-1}\frac{\sum_i (r_{ei} \cdot r_{si})}{\sqrt{\sum_i r_{ei}{}^2}\sqrt{\sum_i r_{si}{}^2}}$$

Experiments have been carried out to compare the performance of PC systems with various implementations. Four types of data set called 'Full', 'Big', 'Medium' and 'Small' were used. The Full set is the whole set of randomness outputs from the above 'randomness scanning' experiment. The other three types are extracted from the Full set. The Big set is comprised of output from files that are over 20 KB. The Medium set is comprised of output from files in the range of 10 KB – 19 KB and the Small set comprises output from files smaller than 10 KB.

Results of the sliding window algorithm need to be pruned into vectors of the same size for characteristic representation. In this

Pruning method (Sliding window)	Distance measure	Data set type	Total files	Positive	False	Identification rate
First	Sum of squared	Full	691§	561	130	81.19%
	Cosine	Full	691§	572	119	82.78%
Smallest	Sum of squared	Full	691§	578	113	83.65%
	Cosine	Full	690†	639	51	92.61%
Ordered smallest	Sum of squared	Full	691§	624	67	90.30%
	Cosine	Full	690†	676	14	97.97%
Trunk	Sum of squared	Full	693§	662	31	95.53%
	Cosine	Full	693§	686	7	98.99%

Table 1: Evaluation of two distance measures and four pruning strategies using the sliding window algorithm.

§The full data set has 696 packed files. However, some small packed files which produce less than *N* values were removed from the set.

†In the Morphine packed file collection, 'make.exe' was removed since it produces more than 100 of the same smallest values, which are all rescaled to 0.

Algorithm (Pruning method)	Data set type	Total files	Positive	False	Identification rate
Fixed sample count	Full	696	396	300	56.90%
	Big	265	195	70	73.58%
	Medium	236	210	26	88.98%
	Small	185	162	23	87.57%
Sliding window (Trunk)	Full	693*	686	7	98.99%
	Big	263	261	2	99.24%
	Medium	236	234	2	99.15%
	Small	185	184	1	99.46%

Table 2: Packer classification results using both the fixed sample count algorithm and the sliding window algorithm.

paper, the effects of different pruning strategies are compared. For all four pruning methods described above, the system uses 100 low randomness values of the file, that is, *N* is set to 100. These values are chosen as a compromise between the information extracted from the small file and big file.

Tables 1 and 2 illustrate the results of experiments. The Cosine distance measure outperforms the SSD measure in all classification experiments as a distance measure. Among four pruning heuristics, the Trunk pruning is the most effective one followed by the Ordered smallest pruning. This is probably due to the fact that most packers store code at the beginning or the end of the file. This pattern is also displayed in the randomness scanning plot shown in the last section.

Though the overall performance of the fixed sample count algorithm is unsatisfactory, Table 2 shows that packer classification can be achieved by computing the randomness of the file using the sliding window algorithm with the Trunk pruning algorithm. Moreover, the performance can be improved using a categorized data set. It is also noted that with the fixed sample count algorithm, the feature of low randomness is masked or disappears in the big files with small sample count or over-represented in the small files with big sample count. Therefore, adjusting sample count regarding the file size is possible to improve its performance.

Packer classification experiments were also carried out on real malware samples using the signatures generated from the above training data set. For each of six packers, five malware samples which have been detected packing with the specific packer were randomly chosen from the *VET* database[3]. The detail of malware samples is listed in Table 3.

As shown in Table 4, only 24 out of 30 files were classified as having the correct packer using the sliding window algorithm. However, 100% of the small files were correctly classified, though the detection rate for big files is only 75%. It is noted that most falsely detected files are large, and are poorly matched with the file sizes used to create the training data.

Unpacking animation

The randomness test algorithm is also useful for investigating unpacking behaviour by monitoring the memory changes caused by memory writing, decompression and decryption. This is achieved by placing breakpoints on main loops during unpacking, dumping memory, and then performing the detail-preserving randomness analysis on the dump.

Experiments were carried out on five packers: ASPack 2.12, FSG 2.0, MEW 11, Morphine 2.7 and UPX 2.03w. These animations visualize each packer's unpacking routine. By viewing an unpacking animation, anti-virus researchers can easily get a feel about how a packer is operating. For example, when the uncompressed code is written into the empty section created by the loader, that section's randomness value will change from the lowest to a high value.

The dumping tool, *Multi-dumper*, described here is a C++ plug-in for *IDA Pro* (*IDA*) [10]. This tool allows the user to

[3] This database is the property of *CA*.

Packer name	Sample name	Malware name	Malware type	Data set type
FSG	wrwfl.exe	Win32/Stresid.AU	Trojan	Small
	bjflvgq.exe	Win32/Stresid.AW	Trojan	Small
	Ravdm.exe	Win32/Mirpat.E	Trojan	Big
	Ravdm.exe	Win32/Mirpat.F	Trojan	Big
	OUTPUT.EXE	Win32/SillyDl.CTG	Trojan	Small
MEW	photo.exe	Win32/Robknot.CB	Worm	Big
	inetinfo.exe	Win32/Robknot.CD	Worm	Big
	csrss.exe	Win32/Yurist.O	Worm	Big
	csrss.exe	Win32/Yurist.Q	Worm	Big
	mswsus.exe	Win32/Cuebot.M	Worm	Small
Morphine	hct.exe	Win32/Modtool.Z	Trojan	Big
	devgt.Vexe	Win32/QQPass.DU	Trojan	Big
	doom3d.exe	Win32/Rbot.GKW	Worm	Big
	mv.exe	Win32/Spybot.MX	Worm	Big
	crss.exe	Win32/Spybot.PS	Worm	Big
RLPack	svshosts.exe	Win32/Rbot.GMS	Worm	Big
	unsecapp32.exe	Win32/Rbot.GMU	Worm	Big
	winamp.exe	Win32/Linkbot.FU	Worm	Big
	csrs.exe	Win32/Linkbot.BK	Worm	Big
	ctech.exe	Win32/Mytob.NR	Worm	Big
UPack	tool.exe	Win32/Tonveck.H	Virus	Small
	CalcHash.exe	Win32/Kowird.A	Worm	Big
	load.exe	Win32/SillyDl.DEG	Trojan	Small
	gx.jpg	Win32/Mansund.A	Trojan	Big
	14JMmmCN.exe	Win32/Mansund.A	Worm	Big
UPX	winfgt.exe	Win32/Spranuf.C	Worm	Big
	drsss.exe	Win32/Spranuf.D	Worm	Big
	svichosst.exe	Win32/Nuqel.F	Worm	Big
	SVICHOOST.exe	Win32/Nuqel.G	Worm	Big
	foto.exe	Win32/Temrisc.E	Worm	Big

Table 3: Malware sample set.

Data set type	Total files	Positive	False	Detection rate
Full	30	24	6	80.00%
Big	24	18	6	75.00%
Small	6	6	0	100.00%

Table 4: Malware packer classification result using the sliding window algorithm with the Trunk pruning method.

set breakpoints on the loops. When the program is running in an *IDA* debugger, every time the breakpoint is triggered, the memory of the program is automatically dumped. Interesting loops are identified using the 'hump-and-dump' method [11]. Since an unpacking/decryption or copy loop normally involves a massive recurrence, it will generate a 'hump' in an ordered address execution histogram and be easily identified.

Once dumps are collected, a randomness test described before is applied on each file. Using these randomness values, a *Mathematica* notebook is used to generate an animation of the randomness distribution during unpacking.

CONCLUSION AND FUTURE WORK

This paper presents a fast randomness test algorithm which is able to preserve the local detail of a packer. Such an algorithm is useful for both comparative measurements and as an investigative tool. Our results show that a packer classification system implemented using this algorithm can successfully identify packers even in real malware samples.

Future work includes further exploration of the algorithm with different values of parameters such as the window size, the skip size and the sample count and developing new effective pruning strategies. For a PC system, a variety of distance measures will be evaluated. Also, a larger training set which includes not only packed clean files but also malware samples needs to be built to improve the performance of the system.

BIBLIOGRAPHY

[1] Bishop, M. Computer Security: Art and Science. Addison-Wesley, 2002.

[2] Shannon, C.E. A Mathematical Theory of Communication. October 1948, Bell Systems Technical Journal, pp.379–423.

[3] Shannon, C.E. Communication Theory of Secrecy Systems. October 1949, Bell Systems Technical Journal, pp.656–715.

[4] Knuth, Donald E. The Art of Computer Programming. 3rd Edition. Addison-Wesley, 2004. Vol. 2.

[5] Marsaglia, G. The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness. Department of Statistics – Florida State University. http://stat.fsu.edu/pub/diehard/.

[6] Brown, R.G. Robert G. Brown's General Tools Page. Duke University Physics Department. http://www.phy.duke.edu/~rgb/General/dieharder.php.

[7] Huffman, D.A. A Method for the Construction of Minimum-Redundancy Codes. 1952, Proc. Inst. Radio Eng, Vol. 40, pp.1098–1101.

[8] Moffat, A.; Turpin, A. Compression and Coding Algorithms. Kluwer, 2002.

[9] Syring, K.M. GNU utilities for Win32. http://unxutils.sourceforge.net/.

[10] Hex-Rays SA. The IDA Pro disassembler and debugger. http://www.hex-rays.com/idapro/.

[11] Sun, L.; Ebringer, T.; Boztas, S. Hump and dump: efficient generic unpacking using an ordered address execution histogram. 2008. 2nd CARO workshop. Available at http://www.datasecurity-event.com/uploads/hump_dump.pdf.