Using Entropy Analysis to Find Encrypted and Packed Malware

In statically analyzing large sample collections, packed and encrypted malware pose a significant challenge to automating the identification of malware attributes and functionality. Entropy analysis examines the statistical variation in malware executables, enabling analysts to quickly and efficiently identify packed and encrypted samples.

Robert Lyda
Sparta

James Hamrock
McDonald Bradley

Malware authors often use encryption or packing (compression) methods to conceal their malicious executables' string data and code. These methods—which transform some or all of the original bytes into a series of random-looking data bytes—appear in 80 to 90 percent of malware samples.[1] This fact creates special challenges for analysts who use static methods to analyze large malware collections, as they must quickly and efficiently identify the samples and unpack or decrypt them before analysis can begin. Many tools, including the packing tools themselves, are generally successful at automatically unpacking or decrypting malware samples, but they're not effective in all cases. Oftentimes, the tools fail to recognize and reverse the transformation scheme or find the original entry point in the malware binary. Many malware samples thus remain packed or fully encrypted, and analysts must identify them for manual analysis and reverse engineering.

The difficulty of recognizing these transformed bytes can vary greatly, depending on the transformation scheme's strength and the original bytes' statistical nature. However, stronger transformation schemes—such as Triple DES encryption—typically produce less predictable sequences. This principle serves as the basis for Bintropy, a prototype binary-file entropy analysis tool that we developed to help analysts conveniently and quickly identify encrypted or packed malware. Bintropy operates in multiple modes and is applicable to any file. Here, we focus on files in the Windows Portable Executable (PE) format, which comprises the format of the majority of malware executables.

Bintropy uses an established entropy formula to calculate the amount of statistical variation of bytes in a data stream. Specifically, it sums the frequency of each observed byte value (00h – FFh) that occurs in fixed-length data blocks, and then applies the entropy formula to generate entropy scores. Higher entropy scores tend to correlate with the presence of encryption or compression. Further, to compensate for the variety of known packing and encryption tools and the varying degree of transformations they produce, we developed a methodology that uses Bintropy to discriminate between native executables and those that have been packed or encrypted. Our methodology leverages the results obtained from training Bintropy over different sets of executable file types to derive statistical measures that generalize each file type's expected entropy ranges. The methodology compares the malware executable's entropy traits—which Bintropy computes—against the expected ranges to determine if the malware is packed or encrypted.

Here, we describe the Bintropy tool and methodology. We also discuss trends associated with malware encryption and packing, which we discovered by applying the tool and methodology to a corpus of 21,576 PE-formatted malware executable files obtained from a leading antivirus vendor.

Approach and technical analysis

Following a description of entropy and its measurement, we describe how we use entropy analysis to identify packed or encrypted malware executables. We then offer results from testing our methodology.

Entropy analysis

Information density, or entropy, is a method for measuring uncertainty in a series of numbers or bytes.[2] In technical terms, entropy measures the level of difficulty or the probability of independently predicting each number in the series. The difficulty in predicting successive numbers can increase or decrease depending on:

- the amount of information the predictor has about the function that generated the numbers, and
- any information retained about the prior numbers in the series.

For example, suppose we had a sequence of *n* consecutive numbers in a Fibonacci series, which is a sequence of numbers computed by adding the successive sums of the preceding two numbers in the series. If we had knowledge of how the Fibonacci function worked or saw enough numbers in the series to recognize the pattern, we could predict the series' next number with absolute certainty. In effect, the entropy changes when the predictor applies prior knowledge or relevant knowledge gained to determine the probabilities of successive numbers in the series. Thus, receiving information about the generator function reduces the entropy by the value of the information received.[3]

Although a sequence of good random numbers will have a high entropy level, that alone doesn't guarantee randomness. For example, a file compressed with a software compressor—such as gzip or winzip—might have a high entropy level, but the data is highly structured and therefore not random.[4] Simply observing entropy will not necessarily provide enough information to let the observer distinguish between encryption and compression unless the observer knows how the data was generated. We can compute the entropy of a discrete random event *x* using the following formula[2]:

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

where *p(i)* is the probability of the i^{th} unit of information (such as a number) in event *x*'s series of *n* symbols. This formula generates entropy scores as real numbers; when there are 256 possibilities, they are bounded within the range of 0 to 8.

Bintropy: A binary entropy analysis tool

Bintropy is a prototype analysis tool that estimates the likelihood that a binary file contains compressed or encrypted bytes. Specifically, the tool processes files by iterating through fixed-length data blocks in the binary, summing the frequency of each block's observed byte values (00h – FFh). From this, it calculates the block's entropy score. In addition to individual block entropy scores, Bintropy calculates other entropy-related file attributes, including the average and highest entropy scores. Finally, Bintropy formulates an overall confidence score by using a rule-based methodology to analyze the block entropy scores against a set of predefined entropy attribute metrics.

Bintropy has two modes of operation. In the first, the tool analyses the entropy of each section of PE-formatted executables, as specified in the executable's header. This helps analysts determine which executable sections might be encrypted or packed. A standard compiler-generated PE executable has standard sections (such as .text, .data, .reloc, .rsrc). However, many packing tools modify the original executable's format, compressing the standard section's code and data and collapsing them into the one or two new sections. In this mode, Bintropy calculates an entropy score for each section it encounters. It doesn't calculate a score for the header section, which in our experience is unlikely to contain encrypted or compressed bytes.

Bintropy's second operational mode completely ignores the file format. Instead, it analyzes the entire file's entropy, starting at the first byte and continuing through to the last. With a PE-formatted file, users can thus analyze the entropy of code or data hidden at the end of a file or in between PE-defined sections (cavities), which is where stealthy file-infecting samples, such as W32/Etap (http://vil.nai.com/vil/content/v_99380.htm), typically hide.

Entropy metrics and confidence-scoring methodology

There are currently hundreds of different packing algorithms, each of which employs popular compression and/or encryption algorithms such as Huffman, LZW, and polymorphism to protect executable files.[5] However, our entropy analysis objective is not to model the transformations of any specific packing or encryption tool. Rather, it is to develop a set of metrics that analysts can use to generalize the packed or encrypted executable's entropy attributes and thus distinguish them from native (nonpacked or unencrypted) ones. As such, our methodology computes entropy at a naïve model level, in which we compute entropy based only on an executable byte's occurrence frequency, without considering how the bytes were produced.

Although a sequence of good random numbers will have a high entropy level, that doesn't in itself guarantee randomness.

Experiments. To develop a set of entropy metrics, we conducted a series of controlled experiments using the Bintropy tool. Our goal was to determine the optimal entropy metrics for native executable files and files con-

Table 1. Computed statistical measures based on four training sets.

DATA SETS	AVERAGE ENTROPY	99.99% CONFIDENCE INTERVALS (LOW TO HIGH)	HIGHEST ENTROPY (AVERAGE)	99.99% CONFIDENCE INTERVALS (LOW TO HIGH)
Plain text	4.347	4.066 – 4.629	4.715	4.401 – 5.030
Native executables	5.099	4.941 – 5.258	6.227	6.084 – 6.369
Packed executables	6.801	6.677 – 6.926	7.233	7.199 – 7.267
Encrypted executables	7.175	7.174 – 7.177	7.303	7.295 – 7.312

taining data transformations produced by encryption and packing algorithms. The experiments consisted of four separate tests, with training data sets for native, compressed, and encrypted executable files, as well as a set for plain text files for additional comparison.

The *native training set* consisted of 100 Windows 32-bit PE executables, which we alphabetically selected from the default "systems" folder on a Windows XP Service Pack 2 OS environment. The *packed training set* represented a diverse set of packing algorithms; we generated these executables by applying UPX (http://upx.sourceforge.net), MEW1.1 (http://northfox.uw.hu/index.php?lang=eng&id=dev), and Morphine 1.2 (www.hxdef.org) packing transformations to three separate copies of the native executables. To generate the *encrypted training* set, we applied Pretty Good Privacy (PGP; www.pgpi.org/doc/pgpintro) file encryption to the native executables.

We also performed a series of tests using different-sized blocks. The tests determined that 256 bytes is an optimal block size. Tests using larger block sizes, such as 512 bytes, tended to reduce the subjects' entropy scores when encryption existed only in small areas of the executable. Our experiments also showed that executables generally contain many blocks of mostly (or all) zero-value data bytes, which compilers commonly generate to pad or align code sections. This technique can greatly reduce an executable's entropy score, because it increases the frequency of a single value. To compensate for this characteristic, we altered Bintropy to analyze only "valid" byte blocks—that is, blocks in which at least half of the bytes are nonzero.

Results. We applied the Bintropy tool to each of the four training sets to compute individual entropy measures for each set's files. We configured Bintropy to process files in 256-byte-sized blocks and to ignore the executables' format. For each file, the tool computed an average entropy score and recorded the highest block entropy score. Using standard statistical measures, we then aggregated the data for each data set, computing each set's respective entropy and highest entropy score averages. For each data set's average and highest entropy scores, we computed a confidence interval—the interval between two numbers (low and high bounds) with an associated probability *p*. We generated this probability from a random sampling of an underlying population (such as malware executables), such that if we repeated the sampling numerous times and recalculated each sample's confidence interval using the same method, a proportion *p* of the confidence interval would contain the population parameter in question.

Using the training data results (see Table 1), we derive two entropy metrics based on the computed confidence intervals for average and highest entropy. Using a 99.99 percent confidence level, executables with an average entropy and a highest entropy block value of greater than 6.677 and 7.199, respectively, are statistically likely to be packed or encrypted. These two values are the lower confidence interval bounds of the entropy measures we computed for packed executables, and form the basis for our methodology for analyzing malware executables for the presence of packing or encryption. If both the Bintropy computed average file entropy and highest entropy block score exceed these respective values, we label a malware executable as packed or encrypted.

Limitations of the approach

It's infeasible to absolutely determine if a sample contains compressed or encrypted bytes. Indeed, Bintropy can produce both false positives and false negatives. False negatives can occur when large executables—those larger than 500kbytes—contain relatively few encrypted or compressed blocks and numerous valid blocks, thereby lowering the executable file's average entropy measure. False positives can occur when processing blocks that score higher than the packing confidence interval's lower bound, but the blocks' bytes contain valid instruction sequences that coincidentally have a high degree of variability.

Using the statistical results computed from our packed training data set, we calculated our confidence-interval-based methodology's expected false positive rate. We treated the packed data sets as a matched pairs t-distribution because the underlying system files were the same and, therefore, the only randomization was that of the packer. We used these intervals as the entropy filter on unknown input samples. We applied a standard t-test to the data set and calculated a Type I error's significance level—in this case, the false positive rate—to be 0.038

percent. This value indicates the likelihood of an unpacked or unencrypted sample passing through the confidence-interval filter. To compute the expected false negative rate, we calculated the statistical likelihood that a packed sample falls outside our 99.99 percent confidence-interval range (that is, within the .01 percent range). Because we're concerned only about packed executables with entropy scores below the interval's lower bounds, we halve this value to obtain the expected false negative rate of .005 percent.

Finally, sophisticated malware authors could employ countermeasures to conceal their encryption or compression use. For example, they could make encrypted bytes generated using strong cryptography look less random by padding the bytes with redundant bytes. They could also exploit our approach by distributing encrypted bytes among invalid blocks, such that the blocks remained invalid. Such countermeasures could reduce the executable's entropy score, and thereby limit Bintropy's effectiveness.

Entropy trends

As mentioned, we identified entropy trends by running Bintropy and applying our confidence-interval methodology to a corpus of 21,567 malware Windows32-based PE executables from a leading antivirus vendor's collection from January 2000 to December 2005. We organized the resulting malware samples by the year and month in which the vendor discovered them.

Our analysis computes the entropy of a malware executable's PE sections and analyzes the trends of packed or encrypted PE sections (as identified by our methodology) across a large time span. With the exception of the header section, we analyzed all sections identified in the malware executables. Further, we performed the analysis without regard to any particular section's purpose or "normal" use. This enabled us to analyze code or data, hidden in or appended to any of the identified sections.

Our trend analysis features the top 13 PE sections that exceeded the packing lower-bound confidence interval threshold in aggregate. These sections comprised eight standard PE sections and five packer-generated sections. The eight standard sections—.text, .data, .rsrc, .reloc, .rdata, .idata, CODE, and DATA—are created by default by most PE-generating compilers. The remaining five sections—.aspack, UPX1, UPX2, pec1, and pec2 are created by packing technologies that replace the default PE-formatted sections and their bytes with custom ones. However, because other packers reuse the default sections, the packers that created the nondefault sections highlighted in our analysis are not necessarily the most prevalent packers in use.

Section packing trends over time

Figure 1 shows which sections were the most often packed or encrypted for a given year. We calculated each section's percentages by totaling the section's encrypted or packed occurrences and dividing that number by the total number of sections that were packed or encrypted that year.

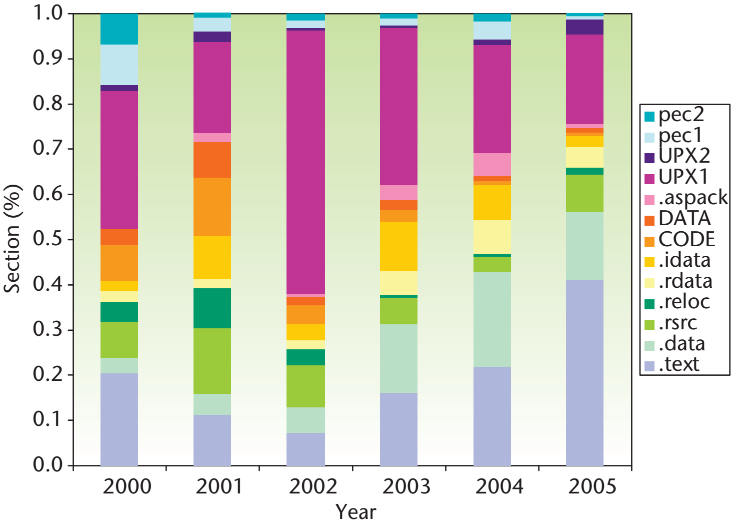

Figure 1. Percentage of encrypted or packed sections over a six-year period. UPX1 was the most prevalent of the packed sections across the period, followed by the .text section.

In 2000, .reloc was the most often packed or encrypted section. This section's popularity steadily declined across the remaining years, with only a slight increase in 2005. The second most-packed section in 2000 was UPX1, which is generated by the very popular UPX packing tool. Due to UPX's prevalent use in numerous W32/GAOBOT and W32/SPYBOT variants, the presence of the UPX1 section in the data set increased over the next few years, peaking in 2002. Thereafter, its prevalence steadily decreased, but UPX1 remained the most popular of all the sections we identified as packed across this six-year period, followed by the .text section, which is where the compiler writes most of a program's executable code.

Evidence of packing in the .rdata section increased in popularity from 2000 through 2005. In 2000 and 2001, packing of the .text, .data, and .rsrc sections was very prevalent, and then decreased in 2002; the sections then steadily increased to peak in 2005. Packing in the CODE, DATA, and .idata sections show no clear trends over the study period. UPX2, pec1, and pec2 were the least-prevalent of the sections we identified as being packed; they were at their lowest in 2003 and were relatively more popular at the time period's beginning and end.

Additional packing trends

Figure 2 shows the annual number of packed or en-

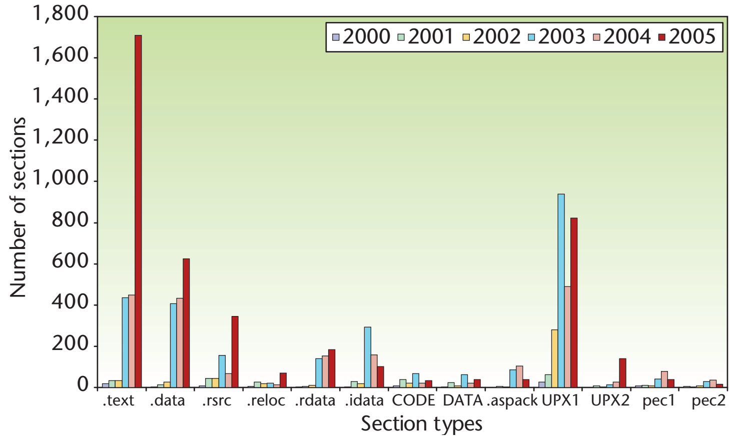

Figure 2. Number of encrypted sections by year. Packing of .text section increased the most dramatically across the period.

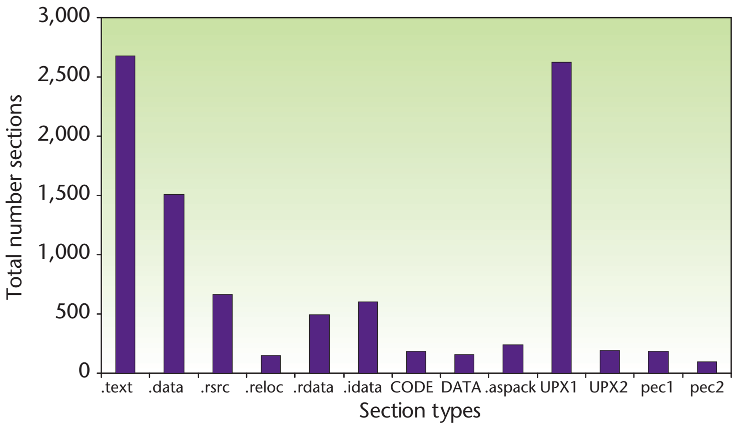

Figure 3. Total number of encrypted sections over the six-year study period. Packing of the .text and .UPX1 were the most prevalent during this time period.

crypted sections for each section type. This graph clearly shows the packing or encrypting trends of particular sections across the years. One of the most notable trends is the increased packing of the .text, .data, .rsrc, and especially the .rdata sections across the period. It also shows UPX1's overall prevalence and the lack of a perceivable trend for the CODE, DATA, and .idata sections.

Figure 3 is an accumulation of Figure 2's data, showing the most commonly packed and encrypted sections over the six-year period. This graph helps delineate each section's exact popularity over the entire period compared to the previous graphs. As Figure 3 shows, the six most commonly packed sections, in order, were .text, UPX1, .data, .rsrc. .idata, and .rdata.

Figure 4 depicts each section's average entropy highscores attribute accumulated over the six-year period. This graph accounts for only those samples that were above the packing confidence interval's lower bound. The overall average entropy value for all other nonpacked sections was approximately 4.1. The graph paints a fairly consistent picture: entropy levels increased from 2000 through 2005 for nearly every section and type of packing/encryption. The exceptions were the DATA section and pec1, which trended up-down-up, and pec2, which trended down. This data indicates that the variability of the bytes that the packing and encryption technologies produced generally increased throughout the six-year period.

Overall, the Bintropy tool proved useful for analyzing and generating statistics on malware collections that contained packed or encrypted samples. It analyzed PE sections of the malware binaries in detail, providing a quick statistical prediction of the existence of significant data randomization, thereby accurately identifying packed or encrypted malware samples. The tool was also successful in identifying encrypted sections and providing statistical data on large-sized malware sample collections at a low level of detail.

The advantage of using entropy analysis is that it offers a convenient and quick technique for analyzing a sample at the binary level and identifying suspicious PE file regions. Once the analysis identifies sections of abnormal entropy values, analysts can perform further detailed analysis with other reverse-engineering tools, such as the IDAPro disassembler.

Our research goal was to develop a coarse-grained methodology and tool to identify packed and encrypted executables. However, a more fine-grained approach might be useful in identifying the particular transformation algorithms that malware authors apply to their malware. To improve Bintropy's entropy computation beyond simple frequency counting, such an approach might further examine the algorithms and the statistical attributes of the transformations they produce to develop profiles or heuristics for fingerprinting their use in malware. □

Acknowledgments

The authors especially thank Jim Horning, David Wells, and David Sames for their technical input and constructive feedback, which helped to significantly improve this article.

References

1. T. Brosch and M. Morgenstern, "Runtime Packers: The Hidden Problem," *Proc. Black Hat USA*, Black Hat, 2006; www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf.
2. C.E. Shannon and W. Weaver, *The Mathematical Theory of Communication*, Univ. of Illinois Press, 1963.

Related work

Using entropy to measure randomness or unpredictability in an event sequence or series of data values is a well-accepted statistical practice in the fields of thermodynamics and information theory.[1,2] In malicious code analysis, researchers have used entropy analysis in various applications. Julien Olivain and Jean Goubault-Larrecq developed the Net-Entropy tool to identify anomalous encrypted network traffic that might indicate a network-based attack.[3]

Closer to our own research, a few tools analyze Portable Executable (PE) file entropy. WinHex (www.winhex.com/winhex/analysis.html) is a commercially available tool that uses entropy to identify common file types, including plain text, jpeg, and binary. Portable Executable Analysis Toolkit (PEAT) is a tool suite that lets analysts examine a Windows PE file's structural aspects.[4] PEAT provides byte-value entropy scores for each PE segment's partitioned section. It then normalizes these entropy values against each window's total entropy. This helps analysts identify section portions that drastically change in entropy value, indicating section-alignment padding or some other alteration of the original file. To use PEAT effectively, analysts must have some domain knowledge about PE files, viruses, and other system-level concepts, as well as some experience working with PEAT.

We've extended PEAT's segment entropy score approach and created a detection tool for automatically identifying encrypted or packed PE executables with a certain degree of confidence. Bintropy has a similar fidelity of analysis capability, but accumulates the information to provide a quick statistical prediction for the existence of significant data randomization, which indicates encryption or packing in large file collections that include executable files. Analysts can also use Bintropy to perform a more in-depth analysis of any particular PE-formatted file section.

Finally, a group of hackers developed and maintains the PEiD analysis tool (http://peid.has.it). According to its Web site, the tool can identify the signatures of more than 600 packing tools. PEiD has an option for analyzing executable files' entropy. However, its developers don't provide detailed documentation on its implementation or underlying methodology.

References

1. C.E. Shannon and W. Weaver, *The Mathematical Theory of Communication*, Univ. of Illinois Press, 1963.
2. R. Clausius, "On the Application of the Theorem of the Equivalence of Transformations to Interior Work," communicated to the Naturforschende Gesellschaft of Zurich, Jan. 27th, 1862; published in the *Viertaljahrschrift of this Society*, vol. vii., p. 48.
3. J. Olivain and J. Goubault-Larrecq, *Detecting Subverted Cryptographic Protocols by Entropy Checking*, research report LSV-06-13, Laboratoire Spécification et Vérification, June 2006; www.lsv.ens-cachan.fr/Publis/RAPPORTS_LSV/PDF/rr-lsv-2006-13.pdf.
4. M. Weber et al., "A Toolkit for Detecting and Analyzing Malicious Software," *Proc. 18th Ann. Computer Security Applications Conf.*, IEEE CS Press, 2002, pp. 423–431.

3. R.W. Hamming, *Coding and Information Theory*, 2nd ed., Prentice-Hall, 1986.
4. M. Haahr, "An Introduction to Randomness and Random Numbers," Random.org, June 1999; www.random.org/essay.html.
5. A. Stephan, "Improving Proactive Detection of Packed Malware," *Virus Bulletin*, 01 Mar. 2006; www.virusbtn.com/virusbulletin/archive/2006/03/vb200603-packed.

***Robert Lyda** is a research engineer at Sparta, where he analyzes malicious code for government and law enforcement agencies. In addition to malware trend and technology assessments, he provides such agencies with detailed reporting of specific malware samples using static and dynamic analysis techniques. His research interests include applying machine-learning mechanisms for classifying malware samples based on statically observable features. He has a BS in computer science from the University of Maryland, College Park. Contact him at robert.lyda@sparta.com.*

***Jim Hamrock** is a software engineer with McDonald Bradley, where he is a leading researcher in malware-analysis trends, applying mathematical and statistical models to study patterns and trends in large sample collections. His research interests include developing algorithms and software analysis tools and reverse engineering of malware samples. He has an MS in applied mathematics from Johns Hopkins University. Contact him at jhamrock@mcdonaldbradley.com.*

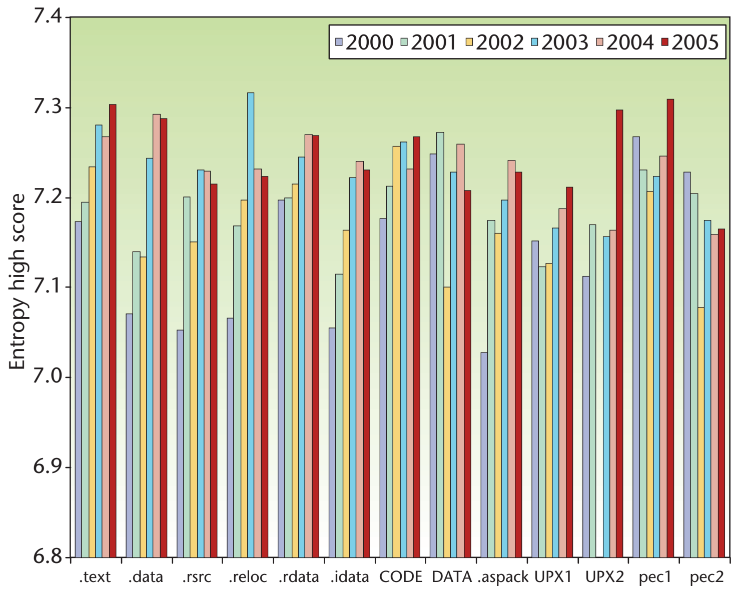

Figure 4. Annual average entropy high scores for each section type. The technologies' strengths generally increased over the study period.