Concordia University College of Alberta

Master of Information Systems Security Management (MISSM) Program

7128 Ada Boulevard, Edmonton, AB

Canada T5B 4E4

The Detection of Operational Malware by its Tactics of Obfuscation

by

RHODES, Don

A research paper submitted in partial fulfillment of the requirements for the degree of

Master of Information Systems Security Management

Date: August 2009

Research advisors:

Dr. Pavol Zavarsky, Director of Research & Associate Professor,

Dr. Dale Lindskog, Assistant Professor, MISSM

The Detection of Operational Malware by its Tactics of Obfuscation

by

RHODES, Don

Research advisors:

Dr. Pavol Zavarsky, Director of Research & Associate Professor,

Dr. Dale Lindskog, Assistant Professor, MISSM

Concordia University College of Alberta
Department of Information Systems Security
7128 Ada Boulevard, Edmonton, Alberta T5B 4E4 Canada

The Detection of Operational Malware by its Tactics of Obfuscation

Submitted in partial completion of the requirements for the degree of
Master of Information Systems Security Management

by Don Rhodes
April, 2009

Advisors:
Dr. Pavol Zavarsky, Director of Research and Associate Professor, ISSM
Dr. Dale Lindskog, Assistant Professor, ISSM

The Detection of Operational Malware by its Tactics of Obfuscation

Don Rhodes
Department of Information Systems Security
Concordia University College of Alberta
7128 Ada Boulevard, Edmonton, Alberta T5B 4E4 Canada

Abstract

While some percentage of new-born malware has always evaded detection by anti-malware services, the quantity of malware able to compromise preventive controls is increasing. Therefore enterprise security practitioners must confront the reality that malware will infect their organization's computing environment. A problem largely unaddressed by the security community is the detection of such operational malware. One aspect of most current malware is powerful techniques of obfuscation which render a malicious payload inscrutable to detectors. Consequently obfuscation serves as a major indicator of operational malware. Various aspects of obfuscation are analyzed with the goal of determining its relevance to the detection process. A differential analysis of various attributes of executables as collected from the disk and memory instances of running malware serve as a basis for evaluating the detective utility of the form of obfuscation commonly called 'packing', that is designed to evade preventive mechanisms before the execution phase. It is established that most malware is detectable by the difference in its code sections between disk and memory; that structural attributes of executables can aid in this detection; and that there are auxiliary obfuscation techniques that must be considered. A tool practicable in the enterprise environment is proposed to remediate this chink in the armor of defensive tactics.

1. Introduction

The quantity of malware able to evade anti-malware services at the time of ingress and successfully infect a target host has been increasing [14]. This increase is due largely to the mechanization of polymorphic and metamorphic obfuscation techniques, so that each instance of malware is unique, reducing the effectiveness of traditional anti-malware detection strategies [3]. Even if anti-malware services have very high rates of detection, the raw volume growth of malware means more instances are undetected and more signatures must be downloaded more often [28]. Commercial anti-malware services must focus on preventing rather than merely detecting infection; must have very low rates of false positives so as not to disrupt legitimate operations; and must do so with a reasonable use of time and resources – a heavy burden in the face of the onslaught of malware variations. Malware authors can rigorously test their 'products' against detective tools, even online (e.g. Virustotal.com, CW Sandbox), so that they can gain assurance that new malware will bypass detection in the general case until signatures are developed. This period of active and unfettered infection has been variously described to have an average duration from days to weeks, but the variance must be large, with the trend towards narrowly targeted payloads leading to longer durations because the anti-malware vendors may be delayed in collecting samples for analysis.

It is this gap in malware defense that this project seeks to address. An enterprise security analyst must ensure the safety of large numbers of workstations and servers, each of which has tens to hundreds of executing processes, any of which may be malware. While sophisticated malware analysis and reverse engineering tools exist, their utility in the enterprise is limited by at least two factors: they are generally heavy weight GUI programs designed for in-depth analysis of a single suspected malware instance; and they offer no help in reducing the size of the set of suspects. The goal of this project is to develop a methodology for identifying the small subset of malware suspects among the larger set of executing processes, so that they can then be more thoroughly analyzed, and remediated if found to be virulent.

Almost all modern malware is obfuscated in order to avert detection by anti-malware services, or to delay the analysis that will result in preventive measures. A malware program is obfuscated if its machine code is somehow scrambled in the image file in order to forestall matching of code signatures at ingress time (e.g. network transfer or installation) and thus detection; or if the program geometry is modified in memory after execution has begun in order to counter reverse engineering activity. Both forms of obfuscation imply subversive intention, and its identification can lead directly to remediation. It is worth pointing out that this project's objective is not in any sense to replace traditional anti-malware services, but rather is dependent on them to maintain the threat of detection that causes malware authors to obfuscate their code. Nor is this methodology a substitute for full intent analysis and reverse engineering. Rather this approach bridges the gap between initially undetected and fully analyzed malware instances by proposing a relatively light weight procedure for identifying suspects and escalating to analysis and remediation.

2. Related Research

In the conclusion of their paper on cryptographic defenses of malware [2], Aycock et al. state that "full analysis of computer viruses may be a luxury of the past" because anti-analysis techniques raise the level of difficulty past any economic threshold. If analysis of the intention of suspected malware cannot feasibly be determined, still intention can be approached via the anti-analysis techniques themselves. The primary tactic for anti-analysis seen in current malware populations is *packing*, a generic term for several types of obfuscation. Packers are available commercially, in open source, and in custom versions [26]. In March 2006, 92% of the malware on the Wildlist (www.wildlist.org) were packed [5, 26]. Innovations such as server-side polymorphism, where malware payloads are packed individually before distribution, make it difficult for signature-based anti-malware defenses to prevent infection [3,5,6,7,21,29]. The alternative to signature-based systems is heuristic analysis, but that tactic is commonly thought to result in an unacceptable rate of false positives [5,26]. However, because this project focuses on operational malware, the false positive bugbear is less of an issue – preventive mechanisms cannot disallow legitimate software, but detective mechanisms have more scope since the installation and execution of the target software has already occurred and because there is a much smaller analysis set.

A foundational technique required by this project is the identification of machine code. The more generic frequencies generated for assembly language instructions and their machine code equivalents accord with published studies [3,30]. The more detailed analyses of particular patterns appear to be idiosyncratic to this project.

Analyzing the payload of potential malware and inferring therefrom its intention, has become difficult for automated anti-malware systems. Disassembly of machine code is useful for intention analysis, as well as normalization for signature-based detection, but anti-disassembly techniques are problematic [12,17,22]. Moser et al. 2007 [17] recommend dynamic code analysis techniques (e. g. emulation) to overcome the disassembly issues they demonstrated with opaque constants. However, the performance of emulators has been unsatisfactory in automatic contexts [13,19,25], and the solution proposed by Graf [11] to overcome performance issues results in an effective emulator but a large footprint that cannot be executed against remote processes. There are two broad approaches to disassembly – linear sweep, a single pass through the code, and recursive traversal, which allows in-depth algorithmic analysis of flow control and thus the identification of non-code areas [22]. The simpler linear sweep will be sufficient for the pattern matching goals of this project.

Rutkowska [20] discussed differential analysis of running processes' memory and the disk image, albeit in the context of rootkit detection. That methodology is used in this project and the technique of identifying machine code is expanded to reveal packing by comparing the disk and memory images. Geometric file analysis, also called shape heuristics, is also considered as a detection mechanism. Many Portable Executable (PE) file attributes [15] are required by the OS to load and execute the program. The TinyPE project has a list of such immutable attributes developed through experiment [24]. These attributes are discussed either in the context of prevention [6,10,16,21,23], or manual Reverse Code Engineering (RCE) [1,8,9], whereas this project's context is chronologically between those two endpoints. Sheehan, et al. (2007) [23] use a large sample set of both malware and known goods (70,000 and 9000 respectively), parse the executable format into measurable attributes, and record the measurements in a database. However, the number of attributes utilized is small, and some of the attributes are 'soft' – i.e. not required to load and execute, and therefore highly mutable by both malware and legitimate programs – e.g. Section Name. This project more systematically analyzes the set of attributes, recording changes made by packers and comparing them to any changes made by the baseline program set. It selects only those attributes that are immutable to reduce false positives, and those which can be conceptualized to be useful to malware authors intent on deterring malware analysis.

3. Methodology

One general assumption for detecting obfuscation is that an analysis of the differences between the executable image file on disk and the image in memory during execution will be revelatory because disk-based code obfuscation must be undone in memory in order for the machine instructions to be executed. A second assumption is that modifications made to structural attributes of an executable image that are critical to the successful loading and initialization of the runtime image, are indicative of analysis obfuscation. A third assumption is that code fragments may be unpacked (de-obfuscated) not *in situ*, but may be removed to unusual locations for execution.

The general methodology is to analyze programs that have been obfuscated or packed by tools that are in common use among authors of both malware and legitimate software protecting intellectual property. The analysis consists of distinguishing attributes that are strong indicators of obfuscation, including those involved in the three assumptions above: the appearance in memory of code fragments not found on disk; changes in geometrical attributes from disk to memory that can be construed as inhibiting analysis/reverse engineering; and the existence of abnormal execution locations. Once the attributes of obfuscation are identified metrics are developed to aid a security practitioner's decision to escalate to full analysis or remediation.

Scope and Constraints

The project was limited to modern Windows 32-bit executables – i.e. the Portable Executable (PE) format; no DOS programs or .NET objects were analyzed. The proposed tool must have a small footprint in order to be useful for bulk operation in an enterprise environment, whether as an installed service, or run ad hoc across the network. This constraint governed some project design decisions.

Data Sources

32 executables were collected for disassembly in order to establish an algorithm for identifying code fragments. Some of these programs were chosen for their supposed similarity to malware – not GUI-based, accessing a wide variety of system resources. Others were chosen in order to round out the sample population and ensure valid statistics for machine code frequencies.

36 programs were collected to serve as a baseline, a sort of control population, for the analysis of what constitutes a normal distribution of structural attributes. This set of programs intersected strongly with the set described above, but was not identical, as a wider range of design paradigms was desired in order to observe as many architectural variations as possible. For example, GoogleEarth was included in this set because it is a huge program that does some interesting things with its code sections.

37 packers were collected for analysis of obfuscation techniques. A single program (CMD.EXE) was packed with these programs – only this one in order to be able to compare techniques among the various packers. CMD.EXE was also chosen for its assumed similarity to console-based malware, and for its ability to remain running while analyzed – many console-based utilities execute a certain function and terminate. One packer would not run a packed cmd.exe (because it did not handle delayed imports correctly), so NOTEPAD.EXE was substituted in that case.

Statistical Analysis of Machine Language

A primary goal of this project was to identify code fragments in memory that did not appear on disk. It follows that code must be identifiable in the general case, and this is not as straightforward as it first appeared, because Intel machine code has a very high entropy – the single byte opcodes cover all 256 possibilities, and the second byte, when an operand, is very dense as well, so that random bits can often be interpreted as opcodes and operands. A frequency analysis identified those opcodes and operands that were far enough from random to be useful in identifying machine code fragments.

There are numerous obstacles to achieving accurate and complete disassembly of machine code [13,17,28] :

- variable length instructions
- indirect addressing mode (requires emulation to know data values)
- opaque constants
- data and code inter-mingled
- strong disassembly requires 2 pass (recursive traversal)

Thus it was decided to avoid dependence on strong disassembly, and focus on purer pattern matching algorithms.

The 32 programs described above were run through a disassembler that generated both machine code and the corresponding assembly language:

```
Address   Machine Code  Assembler
--------- ------------  ---------------------
:4AD0701E 33C0          xor eax, eax
:4AD07020 66A18C3BD34A  mov ax, word[4AD33B8C]
:4AD07026 50            push eax
:4AD07027 8B06          mov eax, dword[esi]
:4AD07029 FF700C        push dword[eax+0C]
:4AD0702C E836AAFFFF    call 4AD01A67
:4AD07031 85C0          test eax, eax
:4AD07033 7505          jne 4AD0703A
```

This allowed frequencies to be calculated for multiple patterns at both the assembly language and machine code levels, using the assembly frequencies to focus on likely candidates in the machine language where the actual identification must occur. For example, the 10 most prevalent assembly constructs:

Count	**Instr**	**%inst**	**%bytes**	**%cum**
369709	mov	27.68	9.06	9.06
241667	push	18.09	5.92	14.98
120441	call	9.02	2.95	17.93
071874	pop	5.38	1.76	19.69
060832	cmp	4.55	1.49	21.18
058132	lea	4.35	1.42	22.60
051245	je	3.84	1.26	23.86
047635	jmp	3.57	1.17	25.02
044987	add	3.37	1.10	26.12
043455	test	3.25	1.06	27.19

led to a more detailed analysis of transfer of control instructions (JUMP, CALL), and test and compare instructions (TEST, CMP). JUMP and CALL are interesting because they include addresses that can drastically reduce the uncertainty (entropy) since the 32bit target address is a small subset of the 2^{32} possibilities of random data, so that e.g. for a FAR CALL there are 6 bytes that are highly unlikely to be seen in a random data set (e.g. `FF15 5634AD40`):

Count	**Instr**	**%inst**	**%bytes**	**%cum**
085558	call XX	6.40	2.10	2.10
019080	jmp XX	1.43	0.47	2.56
008949	je XX	0.67	0.22	2.78
008021	push XX	0.60	0.20	2.98
005810	jne XX	0.43	0.14	3.12
003218	call dword[XX]	0.24	0.08	3.20
001439	mov edi, XX	0.11	0.04	3.24
000977	mov eax, XX	0.07	0.02	3.26
000634	ja XX	0.05	0.02	3.27
000586	jl XX	0.04	0.01	3.29

and so also on the machine code side:

Count	**Instr**	**Opcode**	**%inst**	**%bytes**	**%cum**
042150	je	74	3.16	1.03	1.03
032782	jne	75	2.45	0.80	1.84
024413	jmp	EB	1.83	0.60	2.43
019243	jmp	E9	1.44	0.47	2.90
009089	je	0F84	0.68	0.22	3.13
005853	jne	0F85	0.44	0.14	3.27
003977	jmp	FF	0.30	0.10	3.37

The entropy of JUMP instructions can be further reduced because logic dictates and analysis showed that they are generally preceded by a comparison statement:

```
:4AD02B9D 85C0               test eax, eax
:4AD02B9F 59                 pop ecx
:4AD02BA0 0F852D5E0000       jne 4AD089D3

:4AD06F61 83F801             cmp eax, 001
:4AD06F64 8B8DF0FDFFFF       mov ecx, dword[ebp+FFFFFDF0]
:4AD06F6A 8901               mov dword[ecx], eax
:4AD06F6C 0F8449C60000       je 4AD135BB
```

So, analyze the frequency of comparison statements directly preceding transfers:

Count	**Instr-Pairs**		**%inst**	**%bytes**	**%cum**
022503	test	je	1.68	0.55	0.55
018580	cmp	je	1.39	0.46	1.01
018571	cmp	jne	1.39	0.45	1.46
012043	test	jne	0.90	0.29	1.76
003573	mov	je	0.27	0.09	1.84
002698	mov	jne	0.20	0.07	1.91

and comparisons one or two instructions removed:

	Comparison Instructions Preceding Transfer		
Jump	**0-remove**	**1-remove**	**2-remove**
160091	90112	11048	25331
100%	56%	6.9%	15.8%

These and similar analyses led directly to the machine code identifier algorithm, which is discussed next.

Machine Code Identifier Algorithm

Image sections, formally defined in the PE Format, are searched, both on disk and in memory, for patterns far from random that match machine language. Three stratagems are employed:

1. high frequency 16-bit strings in single instructions, e.g.:
   - 83C4 (ADD ESP); 8B45 (MOV EAX); FF15 (CALL)
2. sequences of instructions within a necessary proximity, e.g.:
   - compare ... jump; push ... call
3. instructions with 32-bit addresses that are relative to the image address space, e.g.:
   - E9 08000000 (Jump forward 8 bytes)

The high frequency16-bit strings are very far from random : for example the two bytes 83C4 constitute .57% of 16-bit strings in code samples, but the expected occurrence for random data would be $(1/256)^2$, or

0.000015%, thus making the most frequent byte pairs excellent signifiers of machine code. Instruction sequences or logical pairs that often occur together, such as TEST ... JE (.55%) and CMP ... JNE (.45%) are counted as signifiers if the comparison instruction is within 20 bytes of the jump instruction. Instructions that use direct 32-bit relative addresses are a rich set of signifiers because they are liberally used by compilers for intersegment references. For example, 2.1% of bytes in a code segment are involved in local procedure calls (FF15 <32-bit addr>). Relative address values are limited to the size of the program image, so that a 1 MB image would have address values that are a tiny fraction of the possible 4 GB range (1/4000 or 0.00025%) making them good signifiers with a low entropy. All 32-bit relative direct addressed jump instructions were matched in order to limit the options of malware authors – so that hand-coded assembly or a modified compiler would be required to avoid such signatures.

These three categories of signifiers are accumulated when matched so that a total count of signifiers is available for each chunk of data (on disk or in memory) analyzed. A chunk size of 4096 bytes was chosen, though the granularity is easily adjusted, and may need to be, as will be discussed below.

Metrics to Identify Code from Signifier Counts

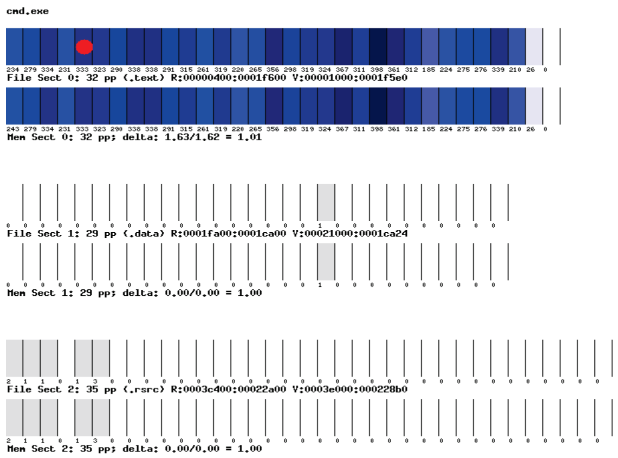
The goal of the project is to identify the appearance of code in the running instance of a program (the process in memory) that does not appear in the storage instance (the file on disk), and thence to infer obfuscation and subversive intent. Although several measurements were tried, the most successful was one of the simplest. For each 4K chunk (or page) in a target data segment, increment a code page index (CPI) if the signifier count exceeded a threshold value. Divide the memory CPI by the file CPI for a target segment to generate a code page delta index (CPDI) whose value will be 1.0 if there is no change in a code segment between disk and memory, and, as experience will show, far greater than 1 (e.g. 28 or 31) when code is unpacked.

Where φ = file and μ = memory:

```
If $\Sigma$ φSig$_i$ > 100 then φCPI += 1
If $\Sigma$ μSig$_i$ > 100 then μCPI += 1
if φCPI == 0 then φCPI++;  μCPI++
CPDI = μCPI / φCPI
```

In order to arrive at this metric, the arrays of code pages were visualized as 'heat maps', with a darker shade of blue indicating more code signifiers, so that large patterns of code segments could be seen, and the accuracy of the metrics verified. In the example below the first program visualized is the standard cmd.exe. It has 3 program sections, each of which is represented in pairs – both its file and memory instances. The first section is quite obviously the code segment, and it can be seen to undergo no changes at run time; nor do the other sections transform at all. This is the standard and expected behavior of legitimate programs. The metrics associated with this program are:

Section	μCPI	φCPI	CPDI
0	30	30	1.00
1	0	0	1.00
2	0	0	1.00



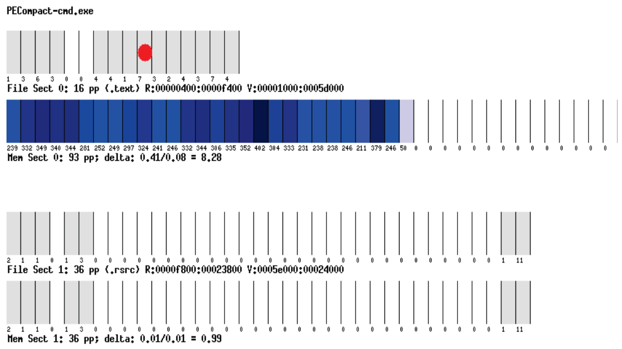
```
cmd.exe
234 279 334 231 333 323 290 338 338 291 315 261 319 220 265 356 298 319 324 367 311 398 361 312 185 224 275 276 339 210 26 0
File Sect 0: 32 pp (.text) R:00000400:0001f600 V:00001000:0001f5e0
243 279 334 231 333 323 290 338 338 291 315 261 319 220 265 356 298 319 324 367 311 398 361 312 185 224 275 276 339 210 26 0
Mem Sect 0: 32 pp; delta: 1.63/1.62 = 1.01

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
File Sect 1: 29 pp (.data) R:0001fa00:0001ca00 V:00021000:0001ca24
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
Mem Sect 1: 29 pp; delta: 0.00/0.00 = 1.00

2 1 1 0 1 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
File Sect 2: 35 pp (.rsrc) R:0003c400:00022a00 V:0003e000:000228b0
2 1 1 0 1 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Mem Sect 2: 35 pp; delta: 0.00/0.00 = 1.00
```

The same program packed by a common obfuscator appears below. There are now only two sections and the first changes significantly (CPDI=28) from file to memory, becoming recognizable code.

Section	μCPI	φCPI	CPDI
0	27	0	28.00
1	0	0	1.00



```
PECompact-cmd.exe
1 3 6 3 0 0 4 4 1 7 3 2 4 3 7 4
File Sect 0: 16 pp (.text) R:00000400:0000f400 V:00001000:0005d000
239 332 349 340 344 281 252 249 297 324 241 246 332 344 306 335 352 402 304 333 231 238 238 246 211 379 246 50 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Mem Sect 0: 93 pp; delta: 0.41/0.08 = 8.28

2 1 1 0 1 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 11
File Sect 1: 36 pp (.rsrc) R:0000f800:00023800 V:0005e000:00024000
2 1 1 0 1 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 11
Mem Sect 1: 36 pp; delta: 0.01/0.01 = 0.99
```

With the intuitive aid of the visual maps the validity of the metric was confirmed, so that large CPDI values were seen exclusively when non-code data on disk became code in memory. It is theoretically possible that a data segment could be transformed from disk to memory, which would require the metrics to distinguish code from data, rather than merely a radical change. The memory CPI metric can be used for this function – if above a certain threshold, the bits in question are code. Because this transformation was not observed, the threshold was not formalized, but a useable value would likely be about 0.33.

Metrics to Identify Geometric Attribute Differences

Most of the geometrical (structural) attributes of the program are utilized by the operating system during the loading and execution of the program image. These attributes must therefore have values reasonable to the OS, and should not change from disk to memory. When changes are observed, it is likely the result of an attempt to deter analysis of the executable. Below are the attributes analyzed, along with some reasonable sample values. These values describe such attributes as the size and location of program data (including code) both on disk and in memory, so that the OS loader can correctly install the program in memory, and the OS resource management functions (e.g. execution scheduler, memory manager) can manipulate the executable as required during its execution phase.

Global Attribute Name	**Value**
e_lfanew	00E0
AddressOfEntryPoint	5056
BaseOfCode	1000
BaseOfData	1f000
ImageBase	4ad00000
SectionAlignment	1000
FileAlignment	0200
SizeOfImage	61000
SizeOfHeaders	0400
LoaderFlags	0000
NumberOfRvaAndSizes	0010

Section Attributes

Name	**3 Value Sets**		
Name	.text	.data	.rsrc
VAddr	1000	21000	3e000
VSize	0001f5e0	0001ca24	000228b0
RAddr	00000400	0001fa00	0003c400
RSize	0001f600	0001ca00	00022a00
Char	0000020	c000004	40000040

These values are compared for each program in the sample set and changes from disk to memory are recorded. Changes are analyzed for intent. The intent of many of the changes was actually experienced – the research tool was unable to analyze the program in memory until the changes were accounted for, protected against. For example, the Vaddr (Virtual Address) attribute indicates the starting location in memory of a section, and should have a 4 to 6 digit hexadecimal value – when the value is something like 454E5245, and the Vsize attribute has a value of 4B006C6C (over 1 GB section size) there is a deliberate attempt to make the analysis program unusable. Another example is the creation of 65,000 bogus sections, which a naive analysis program would attempt to process, resulting in a denial of service situation.

4. Findings

Results are enumerated below and discussed thereafter:

1. None of 36 baseline programs showed any change in their code sections from disk to memory
2. None of 36 baseline programs showed any change in their geometrical attributes from disk to memory
3. 35 of 37 packed samples were unequivocally identifiable as packed by noting a CPDI greater than a threshold value of 10
4. 2 of 37 packed samples did not show significant code changes from disk to memory in their image sections, but further investigation revealed that they were executing their unpacked code from alternate locations and were thus amenable to identification
5. 22 of 37 packed samples modified their geometry with the intent to obfuscate

The confirmation that the baseline programs do not change their code or attributes is important to have a chance to distinguish packed from unpacked, obfuscated from legitimate programs. From this result it is extrapolated that only a very small percentage of legitimate programs obfuscate their code. Such programs are protecting intellectual property or preventing theft, such as computer games, and high-end applications that suppose they have algorithms to keep secret. Both these categories of false positives ought to be easily distinguishable to an enterprise security analyst – the high end applications should be few and known, and games are likely a contravention of policy.

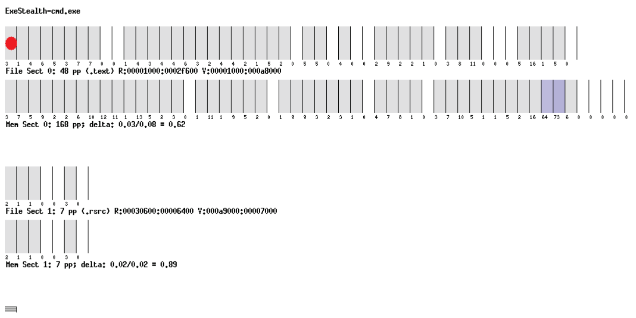
Most (35 of 37) packed samples were clearly identifiable from the appearance in memory of code segments not seen in the disk image. No direct attempt was made by these programs to hide the unpacked code – and indeed, it is difficult to do so, and therein lies the easy success of the detection algorithm. Some sample statistics are listed below with the unpacked code segments in bold italics with asterisks:

Sect	µCPI	φCPI	CPDI	Packer
0 ***	***27***	***0***	***28***	***Armadillo***
1	0	0	1	Armadillo
2 ***	***29***	***0***	***30***	***Armadillo***
3	0	0	1	Armadillo
4	0	0	1	Armadillo
5	0	0	1	Armadillo
6	0	0	1	Armadillo
0 ***	***30***	***0***	***31***	***AsPack***
1	0	0	1	AsPack
2	0	0	1	AsPack
3	0	0	1	AsPack
4	0	0	1	AsPack
4 ***	***11***	***0***	***12***	***ExeCryptor***
5	3	0	4	ExeCryptor
6	0	0	1	ExeCryptor
0	0	0	1	ExeStealth
1	0	0	1	ExeStealth
0 ***	***27***	***0***	***28***	***PrivateExe***
1	0	0	1	PrivateExe
2 ***	***24***	***0***	***25***	***PrivateExe***
3	45	45	1	PrivateExe
4	0	0	1	PrivateExe
0 ***	***30***	***0***	***31***	***RLPAck***
1	0	0	1	RLPAck

Two packed programs did not show code changes upon execution:

Sect	µCPI	φCPI	CPDI	Packer
0	0	0	1	ExeStealth
1	0	0	1	ExeStealth
0	23	23	1	nBinder
1	0	0	1	nBinder
2	0	0	1	nBinder
3	0	0	1	nBinder

In fact, *exestealth* does not show much code at all, which in itself is highly suspicious, albeit difficult to measure:



When detection code was added to the research tool that scanned the executing program's data heaps, executable code was found. Text from the raw analysis file with code page count, location, and size:

```
~Heap: 30 Code Pages Found @00140000->000cb000
~Heap: 18 Code Pages Found @00390000->00047000
~Heap: 30 Code Pages Found @00953000->00020000
```

The *nbinder* program was found to have a child process executing from a temporary location on disk. The detection logic was modified to flag child processes as suspicious if the time of execution is within 3 seconds of the create date of the file, indicating that the program may have unpacked its original disk image into the temporary file rather than its own memory area, thus avoiding the differential analysis of the original algorithm:

```
1 HiddenChild:   C:\Temp\cmd.exe   nBinder
```

This modification assumes that legitimate programs will run legitimate children from their original disk locations, and that new locations indicate an attempt to hide the code.

The above findings were entirely adequate to identify all packers in the sample, so less attention was paid to the geometrical changes. These structural changes were not universal, but highly indicative when present, and could be assigned a weight and factored in to a comprehensive obfuscation index, along with the more pervasive code change metric (CPDI). The intent of these obfuscations is to deter the automated (e.g. by anti-malware services) or manual analysis of the program, thus improving the malware's chances of evading detection. The examples below are extracted from the analysis report, and represent good candidates for detection parameters. The first field shows the number of programs demonstrating the anomalous attribute, the third the name of the modified attribute, and the subsequent fields are actual data values, often expressed as file~mem pairs:

- the code at the entry point differs from disk to mem indicating code manipulation:

```
        15      Attr    AEP Modified
```

- the entry point is outside the image sections, a serious flouting of convention at least. In this particular case, the image header was collapsed so that code was actually in the header. Probably an attempt to prevent reverse engineering:

```
1       Attr    AEP=-1
```

- the entry point field was modified by the program. Since this is used by the loader to pass control to the program, the modification indicates that it was overlaid after execution began:

```
4       Attr    AddressOfEntryPoint
1625~a596       1501~a596
```

- the section count changed in memory, a structural violation:

```
6       Attr    NumberOfSections        2~3
4~255   4~154   1~3     1~3     4~65444
```

- the pointer in the DOS header to the PE header changed. This indicates serious tampering with the image header, for which there can be no legitimate reason:

```
4       Attr    e_lfanew        00e0~10248f
0100~00e0       0090~00e0       0090~00d8
```

- the program had an inordinate number of sections – a clear attempt to disrupt reverse-engineering and malware analysis:

```
1 !Mem+ ExcessiveMemorySections(154)
1 !Mem+ ExcessiveMemorySections(255)
1 !Mem+ ExcessiveMemorySections(65444)
```

- the program has more sections in memory than on disk, indicating significant post-load changes:

```
8       !Mem+   NocorrespondingFileSection
```

- section pointers were either zeroed out or made excessively large in order to disrupt reverse engineering:

```
10      !Ptr    huge
10      !Ptr    zero
```

A standard section might look like this:

```
~Mem-0-Name   *.text
~Mem-0-VAddr  1000
~Mem-0-VSize  0001f5e0
~Mem-0-RAddr  00000400
~Mem-0-RSize  0001f600
~Mem-0-Char   60000020
```

whereas a heavily obfuscated section geometry is recorded below, with the 3 anomalies found listed following the attributes:

```
~Mem-4-Name   msvcrt.d
~Mem-4-VAddr  454e5245
~Mem-4-VSize  4b006c6c
~Mem-4-RAddr  006c6c64
~Mem-4-RSize  2e32334c
~Mem-4-Char   32335245
~Mem-4-!Mem+  No corresponding File Section
~Mem-4-!Ptr   huge  454e5245-4b006c6c > ?-?
~Mem-4-!Mem+  Excessive Memory Sections
(65444)
```

All six of the attributes, from Name through Char, have anomalous values – this program could not have loaded and executed with such values, so the net effect of these obfuscations is to more clearly reveal subversive intent.

5. Implementation Considerations

Detection Evasion and Countermeasures

This project was very successful in detecting obfuscated processes because in general no defense against such detection is attempted – because the detection method is not deployed. If it were a factor in computing environments, how might malware authors actively defend against this class of detection, and how might the methodology counteract these defenses to raise the bar against malfeasance?

If an installed program is protected by a rootkit, it would likely not be visible to the research tool. The scope of this project was to determine obfuscation in otherwise visible processes, so the tool uses standard operating system functions to gather its information, and those functions are compromisable. It would nevertheless be an imposition on attackers to have to include rootkit technology in their deployment package.

A primary avoidance tactic would be to preserve the similarity of the disk and memory images. Two packers in the sample set attempted to do just that – one by running code from the program heap, the other by writing the unpacked code to a temporary file, and executing it from there. The tool already compensates for these evasions. However, there is a technique commonly seen in malware that would be very difficult to mitigate. If a malicious process has sufficient privileges, it can use the Windows API calls *WriteProcessMemory* and *CreateRemoteThread* to inject unpacked code into the execution space (either the process image itself or any of its associated DLLs) of an otherwise innocent process, and thereby preserve its own image similarity. The mitigation of this stratagem would involve expanding the capability of the tool to include scanning of DLLs, and actually scanning all currently executing objects – greatly increasing the load on the tool and reducing its effectiveness. Perhaps it would be more cost effective to search for signs of this technique (e.g. the usage of the two API calls) than to scan all processes in memory. It is still a partial victory to force attackers to employ the injection technique. Security practitioners seldom have the option to avoid an arms race.

Malware authors have adopted sophisticated test strategies in order to ensure that their products will be effective against malware defenses – and this tool, if widely deployed, would undergo such scrutiny. So it must be assumed that the detection algorithm is known to attackers. A second possible tactic would be to create a counterfeit code segment in the disk image with statistically equivalent but semantically meaningless machine instructions, which would be over-written in memory by the desired and unpacked code. This stratagem is possible because the tool currently merely counts signifiers per 4K page, and does not evaluate whether those counts represent the same signifiers in each context (disk and memory). The counteraction would be to refine the pattern matching of the detection algorithm so that it can recognize the order and location of the signifiers in a page. The challenge would be to preserve the algorithm's efficiency.

A third tactic, also depending on knowledge by the attacker, would be the avoidance of the actual signifiers currently used by the detection algorithm. Intel machine language is rich in semantically equivalent instruction sets – indeed, this feature is widely exploited by malware metamorphic techniques. Such evasion is already difficult to do, as the tool employs the top 42 high-frequency 16-bit patterns, 28 test/compare patterns, and many of the 32-bit relative address instructions, including all such jumps. These pattern matching sets can be increased at will, and even including all relative 16-bit jumps is feasible. The malware author would then face a daunting prospect – to code all jumps indirectly (e.g. load the address into a register) or use only short (8-bit) jumps, which allow a range of only +/- 128 bytes. The effective avoidance of all feasible search patterns would reduce a malware author to semantic poverty, depending on a few obscure instructions that must be either hand-coded in assembly language, or generated by a heavily modified compiler. If such a tactic proved to be successful, the very obscurity of the machine code could then be employed in detection.

Deployment

A practical implementation of the tool was envisioned from the start of the project. Although the code was written in Python in order to use the proto-typing features of a high-level language, no object-oriented techniques were used, so that translation to ANSII C would be a straight forward task. A compact executable is required if remote execution is desired – as it may be in an enterprise scenario where a security analyst wishes to investigate suspicious processes in an ad hoc manner. So the tool could be re-coded to have a small footprint, dependent on no resources other than standard Windows components. It should be a console-mode program, eschewing the added weight of a GUI, and should write its parsimonious output to stdout. Thus it will be remotely executable on demand with auxiliary tools such as *psexec*.

The tool's output should be condensed and refined to reach a conclusion about the degree of obfuscation and level of suspicion of the target executable – it should calculate a sort of 'malware quotient', using both the code change and geometrical metrics. Additional features should be considered, such as resource usage enumeration – listing open network ports, disk files, and child processes would sometimes enhance an analyst's decision. There exist other tools that perform this function, but having a single tool that gathers all relevant information reduces the network traffic in the remote execution scenario. Finally, the tool could be installed as a service, and designed to perform its evaluation periodically or at process startup, and communicate to a central server, thus furnishing full-time protection against malware that has evaded prevention systems.

6. Conclusion and Future Research

It is a safe assumption that a substantial fraction of new malware evades anti-malware services that depend primarily on signature-based detection methodologies, so that any given organization will have operational malware in their environment with a lifespan long enough to do some damage. This project has demonstrated the feasibility of a heuristic methodology for detecting program obfuscation, a function which is very difficult for preventive systems to do. The tool designed in this project can supply a light weight and effective host intrusion detection service that adds a dimension to an organization's defense in depth strategy – a modest but real improvement.

There are several avenues that might be explored to extend this research. The testing of the methodology could be improved in two ways: by analyzing actual malware samples rather than just the packers that obfuscate them; and by conducting a scan against a substantial number of computers in an environment protected by standard enterprise security controls. Packers were analyzed rather than actual malware samples mostly in the interest of time – to collect and safely execute real malware would be an intricate project. A major assumption of this project was that the packers analyzed represent the majority of obfuscation techniques that are currently utilized by malware. A second major assumption is that there is operational malware in computing environments. A scan of a substantial number of computers might provide eye-opening evidence.

The technical reach of the methodology could likewise be extended in at least two ways: an analysis of API calls, and an enumeration of reverse-engineering deterrents.

API calls are a primary indicator of intention – the example used earlier is a good case in point: if the functions CreateRemoteThread and WriteProcessMemory are present, it indicates an intention to inject code into a foreign process. API calls that are utilized more often by malware than legitimate programs could be identified, and add to the weight of a remediation decision. Even more pertinent to the methodology developed in this project is the deliberate obfuscation of the API calls, precisely because they declare intention so clearly. The various techniques employed to hide the calls could be explored and possibly added to the obfuscation decision factors. A common tactic is to destroy the import table (a cross reference between local addresses and API function pointers in DLLs) and to insert code that dynamically invokes the APIs (using a function such as GetProcAddress). There are several levels of indirection that can be applied in machine code addressing modes, and these lend themselves to obfuscation. Indirection makes static analysis difficult [17] without rigorous emulation which is very resource-intensive. Nevertheless, there may be signifiers that can be statically analyzed to illuminate this class of obfuscation attempts.

Protection against reverse-engineering is a tactic employed by some malware authors. Some of the more common and simpler techniques should be good indications of the intent to obfuscate. In addition to the geometrical modifications already discussed and included in the methodology, code that detects debuggers and virtual machines (e.g. SIDT and LDT) should prove to be good signifiers.

References

1. Amini, P. & Carrera, E. (2006). Reverse Engineering on Windows. *BlackHat USA Conference, 2006*

2. Aycock, J. et al (2005). Anti-Disassembly using Cryptographic Hash Functions. *15th Annual EICAR Conference 2005*

3. Baumgartner, K. (2007). Storm 2007 – Malware 2.0 Has Arrived. *Virus Bulletin Conference, 2007*

4. Bilar, Daniel (2007). Fingerprinting Malicious Code Through Statistical Opcode Analysis. *International Journal of Electronic Security and Digital Forensics*

5. Brosch, T. & Morgenstern, M. (2006). Runtime Packers: The Hidden Problem? *BlackHat USA Conference 2006*

6. Craig, P. (2006). Unpacking Malware, Trojans and Worms. *Ruxcon, 2006*

7. Christodorescu, M. & Jha, S. (2004). Testing Malware Detectors. *Proceedings of the International Symposium on Software Testing and Analysis (2004)*

8. Eilam, E. (2005). Reversing – Secrets of Reverse Engineering. *Wiley, 2005*

9. Falliere, N. (2007). Windows Anti-Debug Reference. *http://www.securityfocus.com/infocus/1893*

10. Fiñones, R. & Fernandez, R. (2006) Solving the metamorphic puzzle. *Virus Bulletin, March 2006*

11. Graf, T. (2005). Generic Unpacking. *Virus Bulletin Conference, 2005*

12. Kruegel, C., Robertson, W., et al. (2004). Static Disassembly of Obfuscated Binaries. *Proceedings of the* 13th USENIX Security Symposium, 2004

13. Martignoni, L., Christodorescu, M. et al. (2007). OmniUnpack: Fast, Generic, and Safe Unpacking of

Malware. *Proceedings of 23nd Annual Computer Security Applications Conference, 2007*

14. Marx, Andreas (2008). Malware vs. Anti-Malware: (How) Can We Still Survive? *http://www.av-test.org/down/papers/2008-02_vb_comment.pdf*

15. Microsoft (2006). Microsoft Portable Executable and Common Object File Format Specification. *http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx*

16. Mirasw, L. & Steele, K. (2005). Static Malware Detection. *ToorCon, 2005*

17. Moser, A., Kruegel, C., et al. (2007). Limits of Static Analysis for Malware Detection. *Proceedings of 23nd Annual Computer Security Applications Conference, 2007*

18. Quist, D. & Valsmith (2007). Covert Debugging: Circumventing Software Armoring Techniques. *BlackHat USA Conference, 2007*

19. Royal, P., Halpin, M. et al. (2006). PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. *Proceedings of 22nd Annual Computer Security Applications Conference, 2006*

20. Rutkowska, J. (2005). System Virginity Verifier. *http://invisiblethings.org/code.html*

21. Schultz, M., Eskin, E. et al. (2001). Data Mining Methods for Detection of New Malicious Executables. *IEEE Symposium on Security and Privacy, 2001*

22. Schwartz, B, et al. 2002. Disassembly of executable code revisited (2002). *Proceedings of the ninth working conference on reverse engineering*

23. Sheehan, C. et al. (2007). Pimp My PE: Parsing Malicious and Malformed Executables. *Virus Bulletin Conference, 2007*

24. Solar Eclipse. Tiny PE. *http://www.phreedom.org/solar/code/tinype/*

25. Stepan, A. (2006). Improving proactive detection of packed malware. *Virus Bulletin, March 2006*

26. Szappanos, Gabor (2007). Exepacker Blacklisting. *Virus Bulletin, October, 2007*

27. Szor, P. (2005). The Art of Computer Virus Research and Defense. *Addison Wesley, 2005*

28. Trend Micro (2009) Smart Protection Network. *http://itw.trendmicro.com/smart-protection-network/pdfs/SmartProtectionNetwork_WhitePaper.pdf*

29. Yason, M. V.(2007). The Art of Unpacking. *BlackHat USA Conference, 2007*

30. Z0mbie (Year Unknown). Opcode Frequency Statistics. *http://vx.netlux.org/lib/vzo15.html*

Appendices

Appendix 1 – Assembly and Machine Language Statistics

Assembly Instruction Frequency – Percentage of instructions, of bytes in code segment, and running total by bytes

Instr	Instr%	Byte%	CumByte%
mov	27.68	9.06	9.06
push	18.09	5.92	14.98
call	9.02	2.95	17.93
pop	5.38	1.76	19.69
cmp	4.55	1.49	21.18
lea	4.35	1.42	22.60
je	3.84	1.26	23.86
jmp	3.57	1.17	25.02
add	3.37	1.10	26.12
test	3.25	1.06	27.19
jne	2.89	0.95	28.13
ret	2.26	0.74	28.88
xor	1.98	0.65	29.52
sub	1.15	0.38	29.90
inc	1.11	0.36	30.26
and	1.07	0.35	30.61
or	0.87	0.28	30.90
dec	0.72	0.23	31.13
movzx	0.36	0.12	31.25
jle	0.27	0.09	31.34
shr	0.26	0.08	31.42
jl	0.25	0.08	31.50
jc	0.23	0.07	31.57
jae	0.22	0.07	31.65
leave	0.22	0.07	31.72
rep	0.22	0.07	31.79
jge	0.19	0.06	31.85
jbe	0.17	0.06	31.91
neg	0.16	0.05	31.96
shl	0.16	0.05	32.01
movsx	0.16	0.05	32.07
ja	0.15	0.05	32.12
sbb	0.14	0.04	32.16
sar	0.13	0.04	32.21
jg	0.13	0.04	32.25
repe	0.11	0.04	32.28
not	0.10	0.03	32.32
imul	0.10	0.03	32.35
sete	0.08	0.03	32.38
cdq	0.08	0.03	32.40
movsd	0.08	0.03	32.43
repne	0.07	0.02	32.45
fld	0.07	0.02	32.47
setne	0.06	0.02	32.49
fstp	0.06	0.02	32.51
lodsb	0.05	0.01	32.53
idiv	0.04	0.01	32.54
jb	0.04	0.01	32.55
stosd	0.03	0.01	32.56
adc	0.03	0.01	32.57
div	0.03	0.01	32.58
jns	0.03	0.01	32.59
xchg	0.02	0.01	32.60
stosb	0.02	0.01	32.61

Machine Code Frequency – there is a many to many relationship between assembly and machine code instructions. The asterisks represent multiple assembly codes or partial instructions, such as prefixes

Instr	Asmbl	Instr%	Byte%	CumByte%
8B	mov	15.77	5.16	5.16
E8	call	6.64	2.17	7.33
83	arith*	5.44	1.78	9.11
FF	call*	4.62	1.51	10.63
89	mov	4.43	1.45	12.08
8D	lea	4.35	1.42	13.50
50	push	3.41	1.12	14.62
74	je	3.16	1.03	15.65
6A	push	2.67	0.87	16.52
75	jne	2.45	0.80	17.33
68	push	2.38	0.78	18.11
0F	***	2.12	0.69	18.80
85	test	2.05	0.67	19.47
33	xor	1.89	0.62	20.09
56	push	1.86	0.61	20.70
EB	jmp	1.83	0.60	21.30
C3	ret	1.66	0.54	21.84
51	push	1.61	0.53	22.36
E9	jmp	1.44	0.47	22.84
59	pop	1.40	0.46	23.29
53	push	1.32	0.43	23.73
57	push	1.30	0.43	24.15
5E	pop	1.29	0.42	24.58
66	*	1.22	0.40	24.98
3B	cmp	1.21	0.40	25.37
C7	mov	1.15	0.38	25.75
52	push	1.07	0.35	26.10
8A	mov	1.00	0.33	26.43
55	push	0.90	0.29	26.72
5B	pop	0.89	0.29	27.01
5F	pop	0.89	0.29	27.30
C6	mov	0.87	0.29	27.59
80	add	0.79	0.26	27.85
A1	mov	0.66	0.22	28.06
5D	pop	0.66	0.22	28.28
81	arith*	0.65	0.21	28.49
C2	ret	0.61	0.20	28.69
64	*	0.60	0.20	28.89
88	mov	0.59	0.19	29.08
B8	mov	0.56	0.18	29.26
84	test	0.55	0.18	29.44
2B	sub	0.52	0.17	29.61
03	add	0.49	0.16	29.77
F6	arith*	0.43	0.14	29.91
F7	arith*	0.42	0.14	30.05
C1	shift*	0.39	0.13	30.18
39	cmp	0.37	0.12	30.30
F3	rep	0.33	0.11	30.41
A3	mov	0.26	0.09	30.49
40	inc	0.26	0.08	30.58
BA	mov	0.24	0.08	30.66
7E	jg	0.24	0.08	30.73
72	jc	0.23	0.07	30.81
C9	leave	0.22	0.07	30.88

16-bit machine code frequencies

Instr	Instr%	Byte%	CumByte%
83C4	1.74	0.57	0.57
8B45	1.70	0.56	1.12
FF15	1.42	0.46	1.59
85C0	1.28	0.42	2.01
8B4D	1.11	0.36	2.37
6A00	0.97	0.32	2.69
33C0	0.97	0.32	3.01
FF75	0.81	0.26	3.27
8D45	0.75	0.24	3.52
8B55	0.72	0.23	3.75
8945	0.70	0.23	3.98
0F84	0.68	0.22	4.20
8B44	0.63	0.20	4.41
8D4D	0.61	0.20	4.61
8BEC	0.52	0.17	4.78
6A01	0.45	0.15	4.92
0F85	0.44	0.14	5.07
8B4C	0.44	0.14	5.21
8D4C	0.42	0.14	5.35
8BC3	0.41	0.13	5.48
6489	0.40	0.13	5.61
C745	0.37	0.12	5.73
8944	0.35	0.12	5.85
8BC6	0.35	0.11	5.96
8BCE	0.34	0.11	6.07
8D85	0.33	0.11	6.18
83F8	0.33	0.11	6.29
C645	0.32	0.11	6.40

Most Common Assembly Pairs

Instr Pairs		Instr%	Byte%	CumByte%
mov	mov	10.10	3.30	3.30
push	push	7.02	2.30	5.60
push	call	4.89	1.60	7.20
push	mov	3.98	1.30	8.51
mov	push	3.66	1.20	9.70
call	mov	2.94	0.96	10.66
mov	call	2.70	0.88	11.55
pop	pop	2.13	0.70	12.24
lea	push	1.88	0.62	12.86
test	je	1.68	0.55	13.41
mov	cmp	1.46	0.48	13.89
jmp	mov	1.39	0.46	14.35
cmp	je	1.39	0.46	14.80
cmp	jne	1.39	0.45	15.26
je	mov	1.34	0.44	15.69
call	add	1.28	0.42	16.11
push	lea	1.19	0.39	16.50
mov	jmp	1.18	0.39	16.89
mov	lea	1.17	0.38	17.27
pop	ret	1.16	0.38	17.65
mov	test	1.12	0.37	18.01
ret	push	1.07	0.35	18.36
mov	add	1.06	0.35	18.71
jne	mov	1.05	0.34	19.05
mov	pop	1.05	0.34	19.40
call	push	0.98	0.32	19.71
add	mov	0.97	0.32	20.03
call	pop	0.96	0.32	20.35

Frequency of Asssemble Instructions with 32-bit operands

Instr	Instr%	Byte%	CumByte%
call XX	6.40	2.10	2.10
jmp XX	1.43	0.47	2.56
je XX	0.67	0.22	2.78
push XX	0.60	0.20	2.98
jne XX	0.43	0.14	3.12
call dword[XX]	0.24	0.08	3.20
mov edi, XX	0.11	0.04	3.24
mov eax, XX	0.07	0.02	3.26
ja XX	0.05	0.02	3.27
jl XX	0.04	0.01	3.29
jmp dword[9*edx+XX]	0.04	0.01	3.30
jb XX	0.04	0.01	3.31
mov eax, dword[XX]	0.03	0.01	3.32
jg XX	0.03	0.01	3.33
jle XX	0.03	0.01	3.35
jae XX	0.03	0.01	3.35
jmp dword[XX]	0.03	0.01	3.36
jge XX	0.03	0.01	3.37
mov edx, dword[XX]	0.02	0.01	3.38
jmp dword[9*eax+XX]	0.02	0.01	3.39
jbe XX	0.02	0.01	3.40
mov esi, dword[XX]	0.02	0.01	3.40
jmp dword[9*ecx+XX]	0.01	0.00	3.41
mov edi, dword[XX]	0.01	0.00	3.41
mov edx, XX	0.01	0.00	3.42

Frequency of Test & Jump Instruction Pairs

Instr Pairs		Instr%	Byte%	CumByte%
test	je	1.68	0.55	0.55
cmp	je	1.39	0.46	1.01
cmp	jne	1.39	0.45	1.46
test	jne	0.90	0.29	1.76
mov	je	0.27	0.09	1.84
mov	jne	0.20	0.07	1.91
cmp	jc	0.17	0.06	1.97
cmp	jae	0.17	0.05	2.02
cmp	jl	0.16	0.05	2.07
cmp	jle	0.16	0.05	2.13
cmp	ja	0.14	0.04	2.17
cmp	jbe	0.13	0.04	2.21
cmp	jge	0.12	0.04	2.25
sub	je	0.12	0.04	2.29
dec	jne	0.10	0.03	2.33
repe	jne	0.10	0.03	2.36

Appendix 2 – Machine Code Identifier Algorithm

Count various patterns in the data - there are three basic methods:

1. high frequency bit strings in single instructions
2. sequences of instructions (e.g. compare+jump; push+call)
3. instructions with 32-bit addresses that are relative to the image address space, e.g.:
   - E9 08000000 (Jump forward 8 bytes)

   Set up some lookup tables of common bit patterns:

```
H2[0x83C4]=1; H2[0x8B45]=1; H2[0xFF15]=1; H2[0x85C0]=1; H2[0x8B4D]=1;
H2[0x6A00]=1; H2[0x33C0]=1; H2[0xFF75]=1; H2[0x8D45]=1; H2[0x8B55]=1;
H2[0x8945]=1; H2[0x0F84]=1; H2[0x8B44]=1; H2[0x8D4D]=1; H2[0x8BEC]=1;
CC[0x85C0]=1; CC[0x84C0]=1; CC[0x85F6]=1; CC[0x85C9]=1; CC[0x85FF]=1;
CC[0x85DB]=1; CC[0x84C9]=1; CC[0x85D2]=1; CC[0xF645]=1; CC[0xF644]=1;
CC[0x833D]=1; CC[0x837C]=1; CC[0x837D]=1; CC[0x837E]=1; CC[0x83BD]=1;
AD[0xFF]=1;   AD[0x8B]=1;   AD[0x3B]=1;   AD[0x39]=1;   AD[0x83]=1; AD[0x80]=1;
AD[0xC6]=1;   AD[0xC7]=1;   AD[0x3A]=1;   AD[0x8A]=1;   AD[0x3B]=1
```

1. High Frequency double byte patterns:

```
        if H2[buf.W[i/2]]==1: hcnt+=1
```

2. Comparison operators within 20 bytes before a jump instruction:

```
        if buf.B[i] & 0x70 == 0x70:
               jcnt+=getCompare(i,buf)
        def getCompare (q, buf):
               q-=3
               if q % 2 == 1: q-=1
               q/=2
               p=q-20
               if p<0:p=0
               for i in range(q,p,-2):
                      if CC[buf.W[i]]==1:
                             return 1
               return 0
```

3. Calls (E8) and jumps (E9) with 4-byte relative target addresses

```
        if buf.B[i]==0xe8 or buf.B[i]==0xe9:
               memmove(addressof(dw.B),addressof(buf.B)+i+1,4)
               if abs(dw.SL)<size:
                      ecnt+=1;
```

4. All jumps that have nearby comparison operators, or use 4-byte relative addresses:

```
        if buf.B[i]==0x0f and buf.B[i+1] & 0x80 == 0x80:
               jcnt+=getCompare(i,buf)
               memmove(addressof(dw.B),addressof(buf.B)+i+2,4)
               if abs(dw.SL)<size: fcnt+=1; i+=6;
```

5. 1-Byte Move and Push instructions that use 4-byte relative addresses:

```
        if buf.B[i]==0xA1 or buf.B[i]==0xA3 or buf.B[i]==0x68:
               memmove(addressof(dw.B),addressof(buf.B)+i+1,4)
               if (startAddr <= dw.UL < endAddr): dcnt+=1; i+=5;continue
```

6. Various 2-byte instructions that use 4-byte relative addresses:

```
        if (AD[buf.B[i]]==1) and (buf.B[i+1] & 0xCF == 0x05 or buf.B[i+1] & 0xCF == 0x0D):
               memmove(addressof(dw.B),addressof(buf.B)+i+2,4)
               if (startAddr <= dw.UL < endAddr): dcnt+=1; i+=6; continue
```

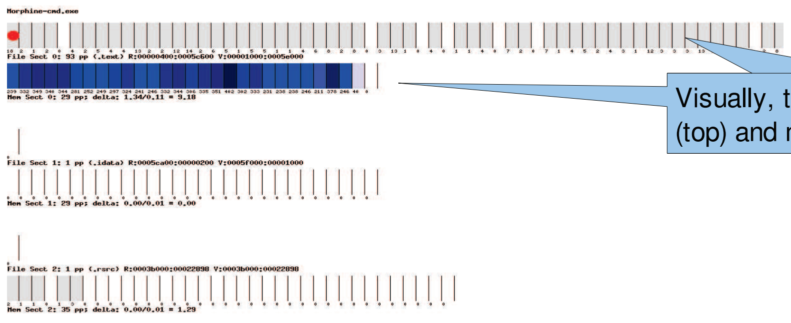
Appendix 3 – Raw Output of Research Tool (per program)

Geometrical Attributes – a heavily packed program; suspect changes marked with asterisks ***

A new section appears in memory – where did it come from?

Change of AEP a major red flag indicating code rewrite and/or entry point obfuscation

```
---------------- Morphine-cmd.exe 2009-02-16 --------
Attrbute                      File    Memory
~e_magic                      5a4d    5a4d
~e_lfanew                     0100    00e0
~Signature                    4550    4550
~Machine                      14c     14c
~NumberOfSections             2       3                    ***
~TimeDateStamp                36881bfc       3b7de326      ***
~PointerToSymbolTable         0000    0000
~NumberOfSymbols              0000    0000
~SizeOfOptionalHeader         e0      e0
~Characteristics              10f     10f
~Magic                        10b     10b
~MajorLinkerVersion           06      07                   ***
~MinorLinkerVersion           53      00                   ***
~SizeOfCode                   5c600   1c800                ***
~SizeOfInitializedData        0000    3f400                ***
~SizeOfUninitializedData      0000    0000
~AddressOfEntryPoint          1625    a596                 ***
~BaseOfCode                   1000    1000
~BaseOfData                   0000    1c000                ***
~ImageBase                    400000  4ad00000             ***
~SectionAlignment             1000    1000
~FileAlignment                0200    0200
~MajorOSVersion               04      05                   ***
~MinorOSVersion               00      01                   ***
~MajorImageVersion            00      05                   ***
~MinorImageVersion            00      01                   ***
~MajorSubsystemVersion        04      04
~MinorSubsystemVersion        00      00
~Win32VersionValue            0000    0000
~SizeOfImage                  60000   5e000                ***
~SizeOfHeaders                0400    0400
~CheckSum                     0000    5cbfc                ***
~Subsystem                    03      03
~DllCharacteristics           00      8000                 ***
~SizeOfStackReserve           100000  100000
~SizeOfStackCommit            10000   100000
~SizeOfHeapReserve            100000  100000
~SizeOfHeapCommit             10000   1000
~LoaderFlags                  0000    0000
~NumberOfRvaAndSizes          0010    0010
~DataDirectory[0]             00000000       00000000      00000000      00000000
~DataDirectory[1]             0005f000       00000200      0001c8c0      00000050      ***
~DataDirectory[11]            00000000       00000000      00000250      00000058      ***
~DataDirectory[12]            00000000       00000000      00001000      000002e4      ***
~DataDirectory[13]            00000000       00000000      0001c564      00000080      ***
```



```
Morphine-cmd.exe
File Sect 0: 93 pp (.text) R:00000400:0005c600 V:00001000:0005e000
Mem Sect 0: 29 pp; delta: 1.34/0.11 = 9.18
File Sect 1: 1 pp (.idata) R:0005ca00:00000200 V:0005f000:00001000
Mem Sect 1: 29 pp; delta: 0.00/0.01 = 0.00
File Sect 2: 1 pp (.rsrc) R:0003b000:00022898 V:0003b000:00022898
Mem Sect 2: 35 pp; delta: 0.00/0.01 = 1.29
```

Visually, the difference between the file section (top) and memory section (bottom) is clear

Code and Section Modifications

```
---------------------------------------------------
~Fil-0-Name    *.text
~Fil-0-VAddr   1000
~Fil-0-VSize   0005e000
~Fil-0-RAddr   00000400
~Fil-0-RSize   0005c600
~Fil-0-Char    e0000020
~Fil-0-stat    E=7     F=2     D=0     H=138   J=273   N=7.95  CP=0
---------------------------------------------------
~Fil-1-Name    .idata
~Fil-1-VAddr   5f000
~Fil-1-VSize   00001000
~Fil-1-RAddr   0005ca00
~Fil-1-RSize   00000200
~Fil-1-Char    c0000060
~Fil-1-stat    E=0     F=0     D=0     H=0     J=0     N=1.02  CP=0
---------------------------------------------------
~Mem-0-Name    *.text
~Mem-0-VAddr   1000
~Mem-0-VSize   0001c7b8
~Mem-0-RAddr   00000400
~Mem-0-RSize   0001c800
~Mem-0-Char    e0000060
~Mem-0-stat    E=3197  F=1884  D=1719  H=217   J=1010  N=6.38  CP=27

File Sect 0
...._......................_..._._..._.._................_
............._..._..............._
Mem Sect 0
@@@@@@@@@@@@@@@@@@@@@@@@@@@-_
---------------------------------------------------
~Mem-1-Name    .data
~Mem-1-VAddr   1e000
~Mem-1-VSize   0001c910
~Mem-1-RAddr   0001cc00
~Mem-1-RSize   0001c600
~Mem-1-Char    e0000060
~Mem-1-stat    E=0     F=0     D=0     H=0     J=0     N=0.19  CP=0

File Sect 1
_
Mem Sect 1
_____________________________
---------------------------------------------------
~Mem-2-Name    .rsrc
~Mem-2-VAddr   3b000
~Mem-2-VSize   00022898
~Mem-2-RAddr   00039200
~Mem-2-RSize   00022a00
~Mem-2-Char    e0000060
~Mem-2-stat    E=2     F=1     D=0     H=4     J=1     N=3.42  CP=0

~Mem-2-!Mem+   No corresponding File Section
File Sect 2
_
Mem Sect 2
..._..______________________________
-----------------------------------------------
~Sec-0 DIF=0.99 CI=0.29/0.00  FI=0.19/0.04   JI=0.87/0.07   Delta: 1.34/0.11=9.18 CP=28.00
~Sec-1 DIF=0.97 CI=0.00/0.00  FI=0.00/0.00   JI=0.00/0.00   Delta: 0.00/0.01=0.00 CP=1.00
~Sec-2 DIF=0.97 CI=0.00/0.00  FI=0.00/0.00   JI=0.00/0.00   Delta: 0.00/0.01=1.29 CP=1.00

~AEP=0

~AEP Modified  71087a0679046683fc285151     508d45e850e8b5fcffff85c0
~Heap: 27 Code Pages Found @4ad00000->0005e000
```

Code page count of first section: 0 for file, 27 for memory. This will result in a CPDI of 28 below, confirming new code in the section (27+1)/(0+1)

New memory section detected

Code change at original entry point. Code also found on the dynamic data heap.

Appendix 4 – Statistical Analysis of Raw Data

Geometrical Attributes – 37 Packed Programs

```
PgmCnt         Attribute      Programs & Values
15      Attr   AEP Modified   ASProtect-cmd.exe     FSG-cmd.exe    Morphine-cmd.exe      ...
1       Attr   AEP=-1         FSG-cmd.exe
12      Attr   AEP=0          ASProtect-cmd.exe     ExeStealth-cmd.exe    Morphine-cmd.exe      ...
4       Attr   AEP=1          bep-cmd.exe    Mew-cmd.exe    RLPAck-cmd.exe upx310-cmd.exe
5       Attr   AEP=2          Expressor-cmd.exe     PESpin-cmd.exe Petite-cmd.exe        ...
15      Attr   AEP=3          Armadillo-cmd.exe     AsPack-cmd.exe MoleBox-cmd.exe       ...
1       Attr   AEP=4          YodaProtector-cmd.exe
1       Attr   AEP=5          ExeCryptor-cmd.exe
4       Attr   AddressOfEntryPoint   1625~a596      1501~a596      1501~5056      5f014~0000     ...
4       Attr   BaseOfData     0000~1c000     64000~69000    2000~1c000     2000~1f000     ...
3       Attr   CheckSum       0000~5cbfc     0000~5cbfc     0000~62494            Morphine-cmd.exe ...
3       Attr   DataDirectory[11]-Size00000250~00000058     00000250~00000058     ...
3       Attr   DataDirectory[12]-Size00001000~000002e4     00001000~000002e4     ...
3       Attr   DataDirectory[13]-Size0001c564~00000080     0001c564~00000080     ...
3       Attr   DataDirectory[1]-Size 0001c8c0~00000050     0001c8c0~00000050     ...
3       Attr   DllCharacteristics    00~8000 00~8000 00~8000        Morphine-cmd.exe      ...
51      Attr   Heap    93 Armadillo-cmd.exe  1 Armadillo-cmd.exe   11 ASProtect-cmd.exe  ...
1       Attr   HiddenChild:   C:\Temp\cmd.exe              nBinder-cmd.exe
4       Attr   ImageBase      400000~4ad00000       4ad00000~0000  1000000~4ad00000 ...
3       Attr   MajorImageVersion     00~05   00~05   00~05          Morphine-cmd.exe      ...
3       Attr   MajorLinkerVersion    06~07   06~07   06~07          Morphine-cmd.exe      ...
3       Attr   MajorOSVersion 04~05   04~05   04~05          Morphine-cmd.exe      SKD-cmd.exe    ...
3       Attr   MinorImageVersion     00~01   00~01   00~01          Morphine-cmd.exe      ...
2       Attr   MinorLinkerVersion    53~00   00~0a          Morphine-cmd.exe      ...
3       Attr   MinorOSVersion 00~01   00~01   00~01          Morphine-cmd.exe      ...
6       Attr   NumberOfSections      2~3     4~255   4~154   1~3     1~3     4~65444        ...
3       Attr   SizeOfCode     5c600~1c800    0a00~1c800     0a00~1f600            ...
2       Attr   SizeOfHeaders  0200~0400      0200~0400             SKD-cmd.exe    ...
1       Attr   SizeOfHeapCommit      10000~1000            Morphine-cmd.exe
4       Attr   SizeOfImage    60000~5e000    7e000~183000   2000~5e000     ...
3       Attr   SizeOfStackCommit     10000~100000   1000~100000    1000~100000           ...
2       Attr   Subsystem      02~03   02~03          SKD-cmd.exe    SkD-Undetectabler-cmd.exe
3       Attr   TimeDateStamp  36881bfc~3b7de326     4683034c~3b7de326     ...
4       Attr   e_lfanew       00e0~10248f    0100~00e0      0090~00e0      ...
--- Sections:
1       !Mem+  ExcessiveMemorySections(154)
1       !Mem+  ExcessiveMemorySections(255)
1       !Mem+  ExcessiveMemorySections(65444)
8       !Mem+  NocorrespondingFileSection
10      !Ptr   huge
10      !Ptr   zero
```

Code and Section Changes – 37 Packed Programs

```
        Sect   Diff   MCode  FCode  Delta  PI     CPDI   Program
        0      1.00   1.34   0.01   452.29 452.29 28.00  Armadillo-cmd.exe
        1      0.97   0.00   0.01   0.00   0.00   1.00   Armadillo-cmd.exe
        2      1.00   0.98   0.13   7.93   7.93   30.00  Armadillo-cmd.exe
        3      0.81   0.12   0.09   1.53   1.24   1.00   Armadillo-cmd.exe
        4      0.66   0.00   0.00   0.50   0.33   1.00   Armadillo-cmd.exe
        5      0.05   0.09   0.10   0.93   0.05   1.00   Armadillo-cmd.exe
        6      0.00   0.00   0.00   1.00   0.01   1.00   Armadillo-cmd.exe
        0      1.00   1.60   0.09   22.25  22.25  31.00  AsPack-cmd.exe
        1      0.97   0.00   0.10   0.00   0.00   1.00   AsPack-cmd.exe
        2      0.89   0.00   0.10   0.00   0.00   1.00   AsPack-cmd.exe
        3      0.00   0.09   0.09   1.00   0.01   1.00   AsPack-cmd.exe
        4      0.00   0.00   0.01   0.00   0.00   1.00   AsPack-cmd.exe
        0      1.00   1.32   0.11   14.04  14.04  28.00  ASProtect-cmd.exe
        1      0.97   0.00   0.05   0.00   0.00   1.00   ASProtect-cmd.exe
        2      0.00   0.00   0.00   0.99   0.01   1.00   ASProtect-cmd.exe
        3      0.08   0.13   0.14   0.97   0.08   1.00   ASProtect-cmd.exe
        4      0.00   0.00   0.01   0.00   0.00   1.00   ASProtect-cmd.exe
        0      1.00   0.41   0.01   188.02 188.02 28.00  bep-cmd.exe
```

```
1       0.00    0.12    0.12    1.01    0.01    1.00    bep-cmd.exe
2       0.89    0.00    0.00    0.24    0.22    1.00    bep-cmd.exe
0       0.00    1.32    1.33    0.99    0.01    1.00    ExeCryptor-cmd.exe
1       0.97    0.00    0.01    0.00    0.00    1.00    ExeCryptor-cmd.exe
2       0.00    0.00    0.00    0.99    0.01    1.00    ExeCryptor-cmd.exe
3       0.00    0.00    0.01    0.00    0.00    1.00    ExeCryptor-cmd.exe
4       1.00    0.66    0.01    157.37  157.37  12.00   ExeCryptor-cmd.exe
5       0.25    0.18    0.08    2.33    0.58    4.00    ExeCryptor-cmd.exe
6       0.00    0.00    0.01    0.00    0.00    1.00    ExeCryptor-cmd.exe
0       0.74    0.03    0.08    0.62    0.45    1.00    ExeStealth-cmd.exe
1       0.00    0.02    0.02    0.89    0.01    1.00    ExeStealth-cmd.exe
0       1.00    0.41    0.01    186.48  186.48  28.00   Expressor-cmd.exe
1       0.00    0.12    0.12    1.00    0.01    1.00    Expressor-cmd.exe
2       0.00    1.34    1.34    1.00    0.01    1.00    Expressor-cmd.exe
3       0.00    0.00    0.00    1.00    0.01    1.00    Expressor-cmd.exe
0       1.00    0.41    0.01    184.93  184.93  28.00   FSG-cmd.exe
1       0.00    0.08    0.08    0.97    0.01    1.00    FSG-cmd.exe
0       1.00    0.41    0.01    184.93  184.93  28.00   Mew-cmd.exe
1       0.23    0.04    0.04    1.21    0.28    1.00    Mew-cmd.exe
0       1.00    1.63    0.08    22.75   22.75   31.00   MoleBox-cmd.exe
1       0.97    0.00    0.02    0.01    0.01    1.00    MoleBox-cmd.exe
2       0.00    0.00    0.00    1.00    0.01    1.00    MoleBox-cmd.exe
3       0.88    1.49    0.25    6.65    5.87    8.00    MoleBox-cmd.exe
4       0.00    0.00    0.01    0.00    0.00    1.00    MoleBox-cmd.exe
5       0.70    0.00    0.09    0.00    0.00    1.00    MoleBox-cmd.exe
0       0.99    1.34    0.11    9.18    9.08    28.00   Morphine-cmd.exe
1       0.97    0.00    0.01    0.00    0.00    1.00    Morphine-cmd.exe
2       0.97    0.00    0.01    1.29    1.26    1.00    Morphine-cmd.exe
0       0.00    1.45    1.44    1.00    0.01    1.00    nBinder-cmd.exe
1       0.00    0.01    0.01    1.02    0.01    1.00    nBinder-cmd.exe
2       0.50    0.04    0.10    0.83    0.41    1.00    nBinder-cmd.exe
3       0.00    0.02    0.02    1.01    0.01    1.00    nBinder-cmd.exe
0       0.00    1.34    1.33    1.01    0.01    1.00    nBinder-HiddenChild-cmd.exe
1       0.00    0.00    0.01    0.00    0.00    1.00    nBinder-HiddenChild-cmd.exe
2       0.00    0.00    0.00    1.00    0.01    1.00    nBinder-HiddenChild-cmd.exe
0       1.00    1.32    0.05    29.77   29.77   28.00   NPack-cmd.exe
1       0.97    0.00    0.01    0.00    0.00    1.00    NPack-cmd.exe
2       0.66    0.00    0.04    0.13    0.09    1.00    NPack-cmd.exe
3       0.00    0.52    0.52    0.00    0.00    1.00    NPack-cmd.exe
0       1.00    0.66    0.01    267.84  267.84  28.00   NsPack-cmd.exe
1       0.03    0.00    0.00    0.97    0.03    1.00    NsPack-cmd.exe
2       0.06    0.10    0.11    0.93    0.05    1.00    NsPack-cmd.exe
0       1.00    0.38    0.01    175.13  175.13  28.00   Obsidium-cmd.exe
1       0.00    0.00    0.01    0.00    0.00    1.00    Obsidium-cmd.exe
2       0.66    0.00    0.03    0.18    0.12    1.00    Obsidium-cmd.exe
3       0.75    0.29    0.12    2.49    1.87    1.00    Obsidium-cmd.exe
0       1.00    0.41    0.01    184.93  184.93  28.00   Orien-cmd.exe
1       0.06    0.09    0.10    0.85    0.05    1.00    Orien-cmd.exe
2       0.00    0.00    0.04    0.08    0.00    1.00    Orien-cmd.exe
3       0.83    0.30    0.06    6.17    5.14    3.00    Orien-cmd.exe
0       1.00    1.34    0.13    12.40   12.40   28.00   Packman-cmd.exe
1       0.97    0.00    0.05    0.00    0.00    1.00    Packman-cmd.exe
2       0.83    0.00    0.04    0.00    0.00    1.00    Packman-cmd.exe
3       0.00    0.03    0.03    0.90    0.01    1.00    Packman-cmd.exe
0       1.00    0.41    0.08    8.28    8.28    28.00   PECompact-cmd.exe
1       0.00    0.01    0.01    0.99    0.01    1.00    PECompact-cmd.exe
0       0.97    1.64    0.10    16.86   16.34   31.00   PECrypt-cmd.exe
1       0.72    0.00    0.43    0.00    0.00    1.00    PECrypt-cmd.exe
2       0.00    0.00    0.00    1.00    0.01    1.00    PECrypt-cmd.exe
3       0.88    0.28    0.44    0.64    0.56    4.00    PECrypt-cmd.exe
0       1.00    1.31    0.06    27.56   27.56   28.00   PELock-cmd.exe
1       0.97    0.00    0.07    0.00    0.00    1.00    PELock-cmd.exe
2       0.66    0.00    0.08    0.06    0.04    1.00    PELock-cmd.exe
3       0.20    0.24    0.20    1.21    0.24    1.00    PELock-cmd.exe
0       1.00    1.32    0.10    15.46   15.46   28.00   PEPack-cmd.exe
1       0.97    0.00    0.01    0.00    0.00    1.00    PEPack-cmd.exe
2       0.00    0.00    0.00    0.99    0.01    1.00    PEPack-cmd.exe
3       0.67    0.04    0.12    0.37    0.24    1.00    PEPack-cmd.exe
0       1.00    1.32    0.08    18.86   18.86   28.00   PESpin-cmd.exe
1       0.97    0.00    0.02    0.00    0.00    1.00    PESpin-cmd.exe
```

```
2       0.00    0.00    0.00    0.99    0.01    1.00    PESpin-cmd.exe
3       0.83    0.47    0.04    11.66   9.72    2.00    PESpin-cmd.exe
0       1.00    0.66    0.09    10.31   10.31   28.00   Petite-cmd.exe
1       0.00    0.00    0.00    0.99    0.01    1.00    Petite-cmd.exe
2       1.00    0.00    0.06    0.00    0.00    1.00    Petite-cmd.exe
0       1.00    1.34    0.06    24.13   24.13   28.00   PolyEnE-cmd.exe
1       0.93    0.00    0.01    0.00    0.00    1.00    PolyEnE-cmd.exe
2       0.57    0.00    0.08    0.06    0.04    1.00    PolyEnE-cmd.exe
3       0.00    0.02    0.02    0.00    0.00    1.00    PolyEnE-cmd.exe
0       1.00    0.41    0.01    218.95  218.95  28.00   PrivateExe-cmd.exe
1       1.00    0.00    0.01    0.00    0.00    1.00    PrivateExe-cmd.exe
2       0.99    0.28    0.01    131.40  130.15  25.00   PrivateExe-cmd.exe
3       0.03    0.16    0.17    0.95    0.03    1.00    PrivateExe-cmd.exe
4       0.00    0.00    0.00    0.99    0.01    1.00    PrivateExe-cmd.exe
0       1.00    0.53    0.01    241.89  241.89  31.00   RLPAck-cmd.exe
1       0.00    0.08    0.08    1.00    0.01    1.00    RLPAck-cmd.exe
0       1.00    0.05    0.01    21.00   21.00   5.00    sdProtector1-notepad.exe
1       0.00    0.10    0.17    0.61    0.01    1.00    sdProtector1-notepad.exe
2       0.22    0.16    0.17    0.93    0.21    1.00    sdProtector1-notepad.exe
3       0.71    0.58    0.08    7.29    5.15    11.00   sdProtector1-notepad.exe
0       1.00    1.34    0.68    6.69    6.69    28.00   SKD-cmd.exe
1       0.97    0.00    0.01    0.00    0.00    1.00    SKD-cmd.exe
2       0.97    0.00    0.01    0.00    0.00    1.00    SKD-cmd.exe
0       1.00    1.63    0.68    8.37    8.37    31.00   SkD-Undetectabler-cmd.exe
1       0.97    0.00    0.01    0.01    0.01    1.00    SkD-Undetectabler-cmd.exe
2       0.97    0.00    0.01    0.00    0.00    1.00    SkD-Undetectabler-cmd.exe
0       1.00    1.32    0.09    17.15   17.15   28.00   swcompress-cmd.exe
1       0.97    0.00    0.05    0.00    0.00    1.00    swcompress-cmd.exe
2       0.00    0.00    0.00    0.99    0.01    1.00    swcompress-cmd.exe
3       0.00    0.04    0.04    0.00    0.00    1.00    swcompress-cmd.exe
4       0.00    0.00    0.01    0.00    0.00    1.00    swcompress-cmd.exe
0       1.00    1.32    0.06    24.13   24.13   28.00   teLock-cmd.exe
1       0.97    0.00    0.07    0.00    0.00    1.00    teLock-cmd.exe
2       0.83    0.00    0.06    0.00    0.00    1.00    teLock-cmd.exe
3       0.00    0.03    0.03    0.82    0.01    1.00    teLock-cmd.exe
0       1.00    0.66    0.09    10.65   10.65   28.00   Themida-cmd.exe
1       0.80    0.00    0.09    0.08    0.06    1.00    Themida-cmd.exe
2       0.00    0.00    0.01    0.00    0.00    1.00    Themida-cmd.exe
3       0.55    0.15    0.08    2.19    1.21    95.00   Themida-cmd.exe
0       1.00    0.41    0.00    1514.95 1514.95 28.00   UPack-cmd.exe
1       0.21    0.05    0.06    0.82    0.17    1.00    UPack-cmd.exe
2       0.00    0.00    0.00    0.00    0.00    1.00    UPack-cmd.exe
0       1.00    0.52    0.01    224.88  224.88  28.00   upx310-cmd.exe
1       0.00    0.01    0.12    0.05    0.00    1.00    upx310-cmd.exe
2       0.00    0.02    0.02    0.89    0.01    1.00    upx310-cmd.exe
0       1.00    0.66    0.06    15.45   15.45   28.00   XComp-cmd.exe
1       0.00    0.00    0.00    0.99    0.01    1.00    XComp-cmd.exe
2       0.00    0.29    0.29    0.00    0.00    1.00    XComp-cmd.exe
0       0.97    1.33    0.05    25.13   24.26   28.00   YodaCryptor-cmd.exe
1       0.00    0.00    0.01    0.00    0.00    1.00    YodaCryptor-cmd.exe
2       0.00    0.00    0.00    1.00    0.01    1.00    YodaCryptor-cmd.exe
3       0.50    0.01    0.00    6.93    3.47    1.00    YodaCryptor-cmd.exe
0       1.00    1.32    0.08    18.40   18.40   28.00   YodaProtector-cmd.exe
1       0.97    0.00    0.01    0.00    0.00    1.00    YodaProtector-cmd.exe
2       0.83    0.00    0.02    0.34    0.28    1.00    YodaProtector-cmd.exe
3       0.80    0.00    0.04    0.18    0.15    1.00    YodaProtector-cmd.exe
4       0.56    0.02    0.17    0.16    0.09    1.00    YodaProtector-cmd.exe
```

Geometrical Attributes – 36 Baseline Programs

```
PgmCnt         Attribute      Programs & Values
36     Attr    AEP=0          AcroRd32.exe   autoruns.exe   calc.exe      ...
18     Attr    Heap    2 AcroRd32.exe 3 googleearth.exe     1 googleearth.exe     ...
--- Sections:
<no data – no anomalies>
```

Code and Section Changes – 36 Baseline Programs

```
       Sect    Diff    MCode   FCode   Delta   PI      CPDI    Program
       0       0.00    0.98    0.87    1.13    0.01    1.00    AcroRd32.exe
       1       0.00    0.01    0.01    0.61    0.01    1.00    AcroRd32.exe
       2       0.00    0.02    0.02    0.00    0.00    1.00    AcroRd32.exe
       3       0.00    0.14    0.14    1.00    0.01    1.00    AcroRd32.exe
       0       0.00    2.31    2.27    1.02    0.01    1.00    autoruns.exe
       1       0.00    0.00    0.00    1.03    0.01    1.00    autoruns.exe
       2       0.50    0.00    0.01    0.06    0.03    1.00    autoruns.exe
       3       0.00    0.04    0.04    1.01    0.01    1.00    autoruns.exe
       0       0.00    0.88    0.87    1.00    0.01    1.00    calc.exe
       1       0.50    0.00    0.00    0.69    0.34    1.00    calc.exe
       2       0.00    0.21    0.21    1.00    0.01    1.00    calc.exe
       0       0.00    0.81    0.81    1.00    0.01    1.00    charmap.exe
       1       0.97    0.06    0.01    21.12   20.50   1.00    charmap.exe
       2       0.00    0.01    0.01    1.00    0.01    1.00    charmap.exe
       0       0.00    0.75    0.74    1.00    0.01    1.00    clipbrd.exe
       1       0.50    0.02    0.01    1.45    0.72    1.00    clipbrd.exe
       2       0.00    0.01    0.01    1.01    0.01    1.00    clipbrd.exe
       0       0.00    1.63    1.62    1.01    0.01    1.00    cmd.exe
       1       0.00    0.00    0.00    1.00    0.01    1.00    cmd.exe
       2       0.00    0.00    0.00    1.00    0.01    1.00    cmd.exe
       0       0.00    1.08    1.07    1.01    0.01    1.00    DUMPSEC.exe
       1       0.00    0.01    0.01    1.22    0.01    1.00    DUMPSEC.exe
       2       0.33    0.00    0.00    1.54    0.51    1.00    DUMPSEC.exe
       3       0.00    0.12    0.12    1.01    0.01    1.00    DUMPSEC.exe
       0       0.00    1.49    1.49    1.00    0.00    1.00    EXCEL.EXE
       1       0.25    0.01    0.01    0.98    0.25    1.00    EXCEL.EXE
       2       0.00    0.01    0.01    1.00    0.01    1.00    EXCEL.EXE
       0       0.00    1.01    0.98    1.03    0.01    1.00    filemon.exe
       1       0.00    0.01    0.01    1.14    0.01    1.00    filemon.exe
       2       0.89    0.00    0.01    0.05    0.05    1.00    filemon.exe
       3       0.00    0.67    0.67    1.01    0.01    1.00    filemon.exe
       0       0.00    0.67    0.67    0.00    0.00    1.00    firefox.exe
       1       0.00    0.00    0.01    0.00    0.00    1.00    firefox.exe
       2       0.00    0.00    0.01    0.00    0.00    1.00    firefox.exe
       3       0.00    0.06    0.06    1.00    0.01    1.00    firefox.exe
       4       0.00    0.00    0.01    0.00    0.00    1.00    firefox.exe
       0       0.00    1.42    1.42    1.00    0.01    1.00    frhed.exe
       1       0.00    0.00    0.00    1.00    0.01    1.00    frhed.exe
       2       0.23    0.00    0.00    0.81    0.19    1.00    frhed.exe
       3       0.00    0.00    0.00    1.00    0.01    1.00    frhed.exe
       0       0.00    0.99    0.99    1.00    0.01    1.00    ftp.exe
       1       0.80    0.00    0.01    0.04    0.03    1.00    ftp.exe
       2       0.00    0.00    0.01    0.00    0.00    1.00    ftp.exe
       0       0.00    1.09    1.09    1.00    0.01    1.00    googleearth.exe
       1       0.00    0.01    0.01    1.00    0.01    1.00    googleearth.exe
       2       0.21    0.06    0.07    0.88    0.18    1.00    googleearth.exe
       3       0.00    0.06    0.06    1.01    0.01    1.00    googleearth.exe
       0       0.00    1.07    1.07    1.00    0.01    1.00    iexplore.exe
       1       0.00    0.00    0.01    0.00    0.00    1.00    iexplore.exe
       2       0.00    0.09    0.09    1.00    0.01    1.00    iexplore.exe
       3       0.00    0.02    0.03    0.00    0.00    1.00    iexplore.exe

  <etc. - no code changes (CPDI always 1.00)>
```