Cryptographic Implementations Analysis Toolkit

v.1.02 (September 2008)

© 2008 – Omar Alejandro Herrera Reyna
(oherrera@prodigy.net.mx)

Overview

The Cryptographic Implementations Analysis Toolkit (CIAT) is a compendium of command line and graphical tools whose aim is to help in the detection and analysis of encrypted byte sequences within files (executable and non-executable). It is particularly helpful in the forensic analysis and reverse engineering of malware using cryptographic code and encrypted payloads.

These tools are improved versions of the prototypes developed during my studies at the University of Bradford (MSc in Forensic Computing, 2005-2006).

License and credits

The tools in the CIAT toolkit make use of several libraries free libraries. Next are the credits, a relation of licenses and a relation of the use of these libraries within each of the tools:

Library	Copyright	License type	CIAT tools using it
Distorm64	Copyright (c) 2003, 2004, 2005, 2006, 2007, 2008, Gil Dabah	Proprietary (redistribution in source and binary forms and free use is allowed under certain conditions). See LEGAL.distorm64.txt	CryptoCodeDetector
Mesa 3-D graphics library	Copyright (C) 1999-2007 Brian Paul	Based on MIT license (redistribution in source and binary forms and free use is allowed under certain conditions). See LEGAL.Mesa3-D.txt	CryptoVisualizer
Glut	Copyright (c) Mark J. Kilgard, 1994, 1995, 1996, 1998.	Proprietary (redistribution and free use is allowed but there is no permission to modify). See LEGAL.glut.txt	CryptoVisualizer

FFTW	Copyright (c) 2003, 2007 Matteo Frigo, Massachusetts Institute of Technology	GNU GPL License version 2	CryptoLocator
CEPHES	Copyright 1984 - 1995 by Stephen L. Moshier	Proprietary (free use is allowed; there is no permission to modify and redistribution is not clear). See LEGAL.cephes.txt	CryptoLocator
FXT	Copyright (c) Joerg Arndt	GNU GPL License version 3	CryptoID

Requirements

Executing distributed binaries

Windows Binaries included in this distribution as well as supporting libraries were compiled using gcc, Mingw and Msys.

Linux binaries were compiled using gcc 4.1.2.

They were tested from command line in machine with Windows Vista Home Premium (32 bits) + SP1 and on Linux Gentoo 2008.0 X86 operating systems.

They should run without problems in any computer with Windows 2000, XP or VISTA 32bitsand any Linux x86 with Mesa3-D, but I cannot guarantee that. If you have problems with these binaries or want to run the programs in other platform you'll need to compile them yourself.

Compiling

Version 1.02 includes standard configuration scripts for Unix like systems. The old Makefile (Makefile.linux32) is still included; if you use Windows I suggest you use MINGW+MSYS.

To compile manually under Windows just get copies of the libraries mentioned in the License and Credits section and follow these instructions:

Tool	Instructions
CryptoCodeDetector	Compile the Distorm64 library. Then compile CCD.c and link it using the Distorm64 library. You will need distorm.h to compile it as well.
CryptoVisualizer	Compile or get binary versions for libraries of Mesa 3-D and Glut: libglu, libglut and libopengl (commonly named as: libglu32, libglut32 and libopengl32). Then compile CV_main.c and link it using these 3 libraries. You will need gl.h, glu.h, glut.h as well. A glut dynamic library (e.g. glut32.dll from http://www.xmission.com/~nate/glut.html) will be needed to execute the program.

CryptoLocator	Compile the FFTW library. Then compile CL_main.c, CL_Tests.c and CL_Includes.h along with the following files from the CEPHES Mathematical Library: const.c, gamma.c, igam.c, mconf.h, mtherr.c, isnan.c and polevl.c (all area available in the "Double" directory). Link the resulting object using the FFTW library. You will need fftw3.h as well.
CryptoID	Compile he FXT library Then compile CID.cc and link it using the FXT library. Note that although CID.cc is in reality a C program, it has to be compiled with a C++ compiler (e.g. using g++), due to the fact that FXT is a C++ library that does C++ style name mangling.

Alternatively, you can now compile and install the libraries mentioned above and simply run ./configure-msys_1.0 under MSYS. Then run 'make' and 'make install´. Note that installing distorm64 is not straightforward (see INSTALL.distorm64.txt).

To compile under Linux based systems type:

./configure
./make
./make install

Tools

The following are detailed descriptions of each of the tools included in CIAT. Please note that all tools will output instructions on how to use them if you simply execute them without any parameters.

CryptoLocator

CryptoLocator allows the detection of pseudo-random byte sequences within mixed content files. It was designed specifically to detect encrypted payloads within malware executables, but it can be used for other purposes, such as detecting encrypted communications or verifying the output quality of closed source encryption software.

This tool implements several randomness test algorithms as defined in [RUK 01]. The algorithms implementations provided with this special publication from NIST (SP800-22) were not used. Instead, they were programmed from scratch using the algorithm definitions from the paper in order to maintain a consistent interface with the main routine and improve readability of the code.

Although [Ruk 01] specifies several randomness tests, not all of them are appropriate for analyzing byte streams within files. Only those deemed appropriate in [Her 06] were implemented, along with the scanning algorithm specified in [HER 06] that identifies possible random byte sequences within files with mixed types of sequences (i.e. random and non-random).

The tool will apply the implemented randomness tests sequentially over the identified string candidates. At each step, it will try to adjust the length of the sequence using the randomness test being tried, and will discard strings that do not pass the test or that are considered to small for the tests to be reliable. The largest minimum substring from all tests that were implemented comes from the Discrete Fourier Transform test, which requires aprox. 125 bytes as a minimum to be reliable; this is the last test applied and previous tests require 13 or 32 bytes as a minimum.

The tool will produce an output similar to this:

```
...
Warning: string at [9899-9900] discarded (< min. test string)
Warning: string at [9905-9942] discarded (p=3.323267e-004)
Warning: string at [11026-11027] discarded (< min. test string)
Warning: string at [17000-17012] discarded (p=1.088630e-003)
RND substring candidate at [6176-6188], with p=1.860293e-002
RND substring candidate at [6577-6613], with p=1.036473e-001
RND substring candidate at [8974-8987], with p=3.525247e-002
RND substring candidate at [9091-9107], with p=1.245516e-002
RND substring candidate at [10414-10432], with p=1.154003e-002
RND substring candidate at [10750-10938], with p=7.803842e-001
RND substring candidate at [11391-15560], with p=5.251710e-001
RND substring candidate at [15854-15923], with p=5.759175e-002
substrings= 8 overflow flag= 0

*** Stage 3 - Serial Test from offset 0 to offset 17408
Warning: string at [6176-6188] discarded (< min. test string)
Warning: string at [8974-8987] discarded (< min. test string)
Warning: string at [9091-9107] discarded (< min. test string)
Warning: string at [10414-10432] discarded (< min. test string)
RND substring candidate at [6580-6613], with p=2.439846e-002
RND substring candidate at [10750-10938], with p=5.320848e-001
RND substring candidate at [11392-15560], with p=5.542693e-001
RND substring candidate at [15854-15923], with p=6.297905e-002
substrings= 4 overflow flag= 0

*** Stage 4 - App. Entropy Test from offset 0 to offset 17408
RND substring candidate at [6580-6611], with p=1.022029e-002
RND substring candidate at [10750-10938], with p=1.000000e+000
RND substring candidate at [11392-15560], with p=4.700875e-001
RND substring candidate at [15854-15922], with p=1.090635e-002
substrings= 4 overflow flag= 0

*** Stage 5 - DFT Test from offset 0 to offset 17408
Warning: string at [6580-6611] discarded (< min. test string)
Warning: string at [15854-15922] discarded (< min. test string)
RND substring candidate at [10750-10938], with p=7.150542e-002
RND substring candidate at [11392-15560], with p=7.707805e-001
substrings= 2 overflow flag= 0
…
```

You can see from the output that offsets of random string candidates are being displayed in decimal, between brackets (take into account tat the first position of a byte in a file is 0, not 1). Test probabilities for acceptance or rejection are shown at the end of the line, when applicable. The Null Hypothesis for these algorithms states that a substring is random; the alternative hypothesis states that strings are not random. The alternative hypothesis is chosen if the probability obtained for the test (p) is less than 0.01, as suggested in [Ruk 01].

If the overflow flag is different from 0 then more than the maximum number of candidates was detected in the file (very rare cases). The current limit for the distributed binary versions is set at 10240 (you can modify this in CL_Includes.h and recompile if needed).

Strings marked with the legend “RND substring candidate” at stage 5 are assumed to have passed all the tests for randomness and are considered strong candidates to be of the following types of data:

- Non-random compressed data
- Encrypted data
- Truly pseudo-random data

To distinguish between the first and the last 2 categories you can use the CryptoID tool.

CryptoCodeDetector

CryptoCodeDetector is a tool specifically designed for analysis of PE executables (only 32 bit executables are currently supported). It was created to locate code references to encrypted payloads within malware (i.e. those identified by CryptoLocator in executable files), but it can be used also to search for any reference to any location within a range in the file being analyzed.

Basically, it analyses the PE header (it shows some information of it as well) and then starts disassembling the code section with the distorm64 library. Any references to memory locations corresponding to the offset range in the file specified in the parameters will be shown in assembly language (Intel notation).

Note that the tool is not intended to replace a full disassembler or debugger. It is just a mere aid to quickly locate references to data ranges.

An example of the output follows:

```
C:\ >CryptoCodeDetector.exe test.tst 10752 10938
CryptoCodeDetector v1.0-(c) 2008 Omar A. Herrera Reyna (oherrera@prodigy.net.mx)

* PE32 Header found
* Code Entry Point RVA: 4720 (0x00001270)
* Offset of Entry Point in file: 1648 (0x00000670)
* Entry point is located in section: .text
* Location (offset) of Watchdog in file: [10752,10938] ([0x00002A00,0x00002ABA])

* Watchdog start is located in section: .data
* Base of Image for RVAs (iBase) is: 0x00400000
* Watchdog start RVA + Image Base: 0x00404000
* 5 sections detected:
    Section:     .text  at [1024,10751] ([0x00000400,0x000029FF])
    Section:     .data  at [10752,11263] ([0x00002A00,0x00002BFF])
    Section:    .rdata  at [11264,16383] ([0x00002C00,0x00003FFF])
    Section:      .bss  at [0,0] ([0x00000000,0x00000000])
    Section:    .idata  at [16384,17407] ([0x00004000,0x000043FF])
* Watchdog reference found:
*   fileOffs |iBase+RVA : <opcode> [param.]    <mnemonic> [operands]
   0x0000076A|0x0040136A:  c685 f1feffff a5     MOV BYTE [EBP-0x10f], 0xa5
   0x00000771|0x00401371:  c685 f2feffff 6c     MOV BYTE [EBP-0x10e], 0x6c
   0x00000778|0x00401378:  c685 f3feffff 95     MOV BYTE [EBP-0x10d], 0x95
   0x0000077F|0x0040137F:  c685 f4feffff ce     MOV BYTE [EBP-0x10c], 0xce
   0x00000786|0x00401386:  c685 f5feffff 55     MOV BYTE [EBP-0x10b], 0x55
   0x0000078D|0x0040138D:  c685 f6feffff 25     MOV BYTE [EBP-0x10a], 0x25
   0x00000794|0x00401394:  c685 f7feffff e0     MOV BYTE [EBP-0x109], 0xe0
   0x0000079B|0x0040139B:  c685 e0feffff 0f     MOV BYTE [EBP-0x120], 0xf
   0x000007A2|0x004013A2:  c685 e1feffff 20     MOV BYTE [EBP-0x11f], 0x20
   0x000007A9|0x004013A9:  c685 e2feffff 6a     MOV BYTE [EBP-0x11e], 0x6a
   0x000007B0|0x004013B0:  c685 e3feffff 04     MOV BYTE [EBP-0x11d], 0x4
   0x000007B7|0x004013B7:  c685 e4feffff 10     MOV BYTE [EBP-0x11c], 0x10
   0x000007BE|0x004013BE:  c685 e5feffff 0d     MOV BYTE [EBP-0x11b], 0xd
   0x000007C5|0x004013C5:  c685 e6feffff 6e     MOV BYTE [EBP-0x11a], 0x6e
   0x000007CC|0x004013CC:  c685 e7feffff 08     MOV BYTE [EBP-0x119], 0x8
   0x000007D3|0x004013D3:  8d8d c8edffff        LEA ECX, [EBP-0x1238]
 /------------------------
|   0x000007D9|0x004013D9:  ba 00404000          MOV EDX, 0x404000
 \------------------------
   0x000007DE|0x004013DE:  b8 be000000          MOV EAX, 0xbe
   0x000007E3|0x004013E3:  894424 08            MOV [ESP+0x8], EAX
   0x000007E7|0x004013E7:  895424 04            MOV [ESP+0x4], EDX
   0x000007EB|0x004013EB:  890c24               MOV [ESP], ECX
   0x000007EE|0x004013EE:  e8 dd1f0000          CALL 0x4033d0
   0x000007F3|0x004013F3:  c74424 08 be000000   MOV DWORD [ESP+0x8], 0xbe
   0x000007FB|0x004013FB:  c74424 04 00000000   MOV DWORD [ESP+0x4], 0x0
   0x00000803|0x00401403:  8d85 28ffffff        LEA EAX, [EBP-0xd8]
   0x00000809|0x00401409:  890424               MOV [ESP], EAX
   0x0000080C|0x0040140C:  e8 ff1f0000          CALL 0x403410
   0x00000811|0x00401411:  c70424 10270000      MOV DWORD [ESP], 0x2710
   0x00000818|0x00401418:  e8 33030000          CALL 0x401750
   0x0000081D|0x0040141D:  83ec 04              SUB ESP, 0x4
   0x00000820|0x00401420:  8d45 f4              LEA EAX, [EBP-0xc]
   0x00000823|0x00401423:  890424               MOV [ESP], EAX
   0x00000826|0x00401426:  e8 d51f0000          CALL 0x403400
```

Note that when using ranges from CryptoLocator it might be the case that the edges detected by the tool differ (by a few bytes) from the real pseudo-random string. In such circumstances it might also be the case that strings cross section borders. In those situations, CryptoCodeDetector will give you a warning and suggest you to adjust the length so that the string falls within a valid section. Example:

```
C:\>CryptoCodeDetector.exe test.tst 10750 10938
CryptoCodeDetector v1.0-(c) 2008 Omar A. Herrera Reyna (oherrera@prodigy.net.mx)

* PE32 Header found
* Code Entry Point RVA: 4720 (0x00001270)
* Offset of Entry Point in file: 1648 (0x00000670)
* Entry point is located in section: .text
* Location (offset) of Watchdog in file: [10750,10938] ([0x000029FE,0x00002ABA])

* Watchdog start is located in section: .text
    Warning, watchdog start is too close to end of section!
    Use the start offset of next section instead.
* Base of Image for RVAs (iBase) is: 0x00400000
* Watchdog start RVA + Image Base: 0x004035FE
* 5 sections detected:
    Section:     .text  at [1024,10751] ([0x00000400,0x000029FF])
    Section:     .data  at [10752,11263] ([0x00002A00,0x00002BFF])
    Section:    .rdata  at [11264,16383] ([0x00002C00,0x00003FFF])
    Section:      .bss  at [0,0] ([0x00000000,0x00000000])
    Section:    .idata  at [16384,17407] ([0x00004000,0x000043FF])
```

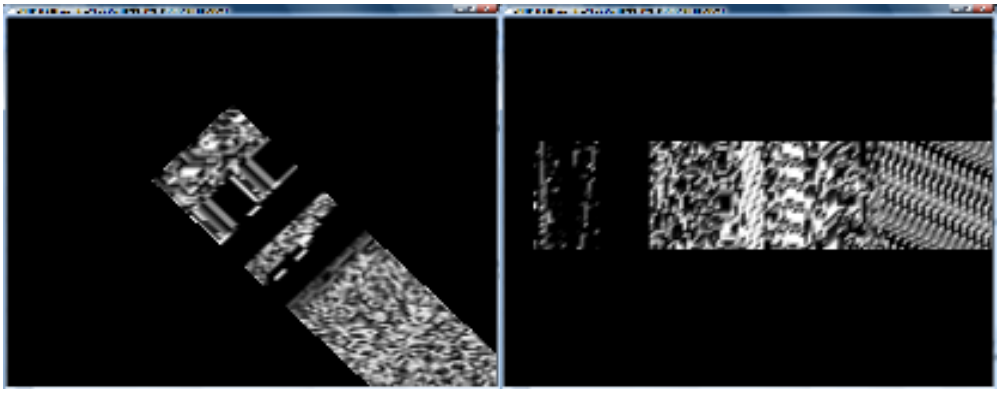
The tool will not stop at the first match. It will continue to report reference matches until it finishes disassembling the code section.

CryptoVisualizer

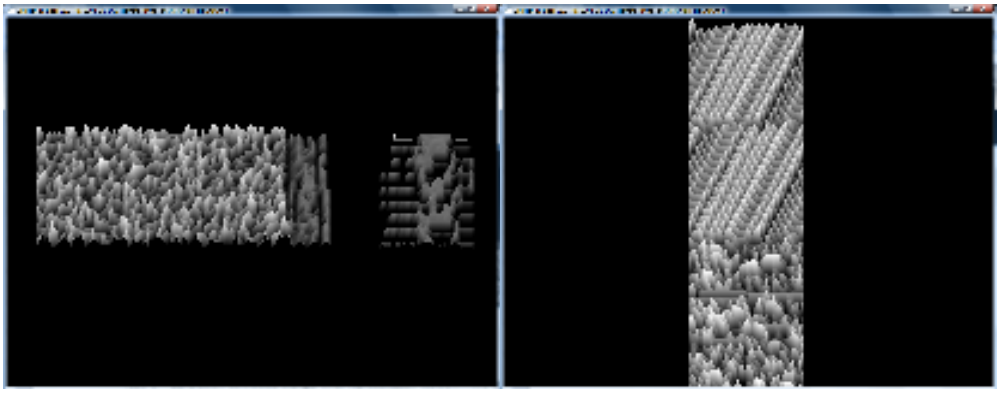
This is a very simple interactive tool that allows you to see graphically any file. It was designed to aid human researchers to quickly pinpoint probably pseudorandom sequences. Of course, this is not a robust technique, but it helps. Source code, executable code, text and other forms of non pseudo-random content in files are also easily identifiable with this tool since bytes forming that kind of sequence tend to fall within a certain range or form visually identifiable patterns.

CryptoVisualizer uses OpenGl style triangle strips and quad strips to represent data, both in 2D and 3D. It supports 5 visualization modes with varying sizes for width and wave amplitude (for 3D representations):

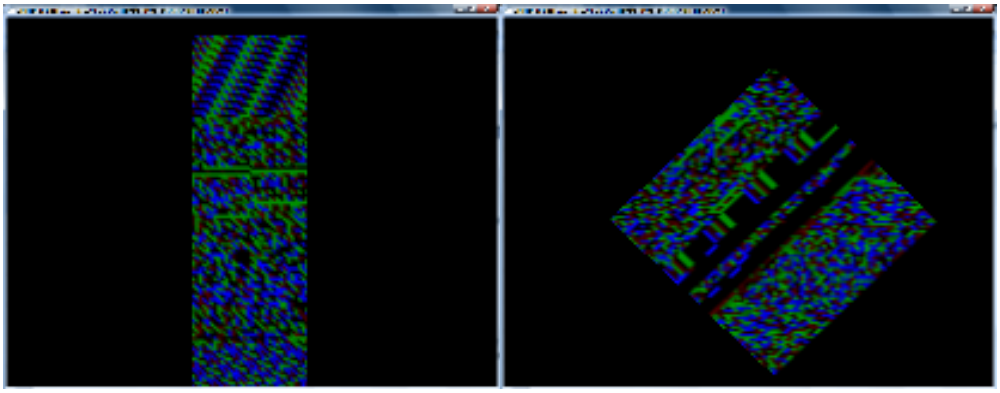
Type 1 - 2D grayscale, connected triangle strip, representations:



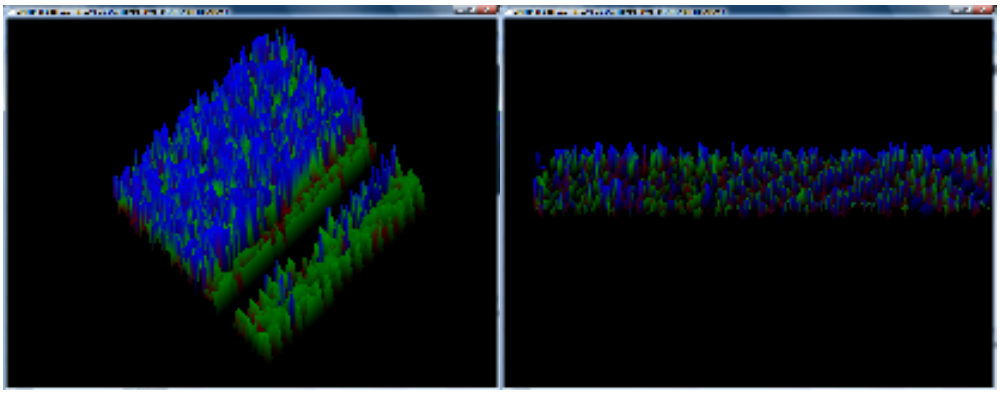
Type 2 – 3D grayscale, connected triangle strip, representations:



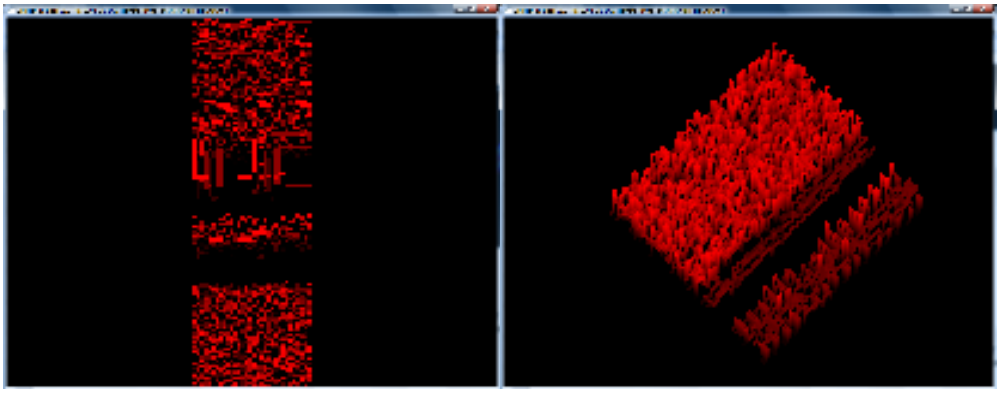
Type 3 – 2D 3-color, connected triangle strip, representation (bytes >= 170 are blue, bytes < 170 and >= 85 are green, bytes < 85 are red):



Type 4 – 3D 3-color, connected triangle strip, representation:



Type 5 – 3D redscale, disconnected quad strip, representations:



Note that you will get a text window with an output log of commands. Most of the time it falls behind the main (graphic) window while opening the application. The current offset of the file, left, top-left or top of the graph will be indicated there, so be sure to move that window to a visible area if you want to take a look at it.

CryptoID

CryptoID is a tool that implements two algorithms: one that assesses the degree of randomness and one that assesses the degree of encryption (for byte sequences that are found to be random). Both are defined in [HER 06] and are based on statistics from Fractional Fourier Transforms applied to the byte sequences being analyzed.

This tool differs from CryptoLocator in that it is not designed to analyze mixed byte sequences (i.e. sequences that contain both pseudo-random and non random subsequences). Therefore, it should be applied only to byte sequences previously detected by CryptoLocator.

The size of the sequence also plays an important role in this technique. Small sequences (less than 300 bytes) tend to produce false positives; same case with long sequences (more than 10000 bytes).

Other than that, the results seem to be accurate but remember this algorithm is new, so take its results with some caution.

Examples of output:

```
C:\>CryptoID.exe test.tst 0 300
CryptoID v1.0 - (c) 2008 Omar A. Herrera Reyna (oherrera@prodigy.net.mx)
Median(Modulus(FrFt(Sequence,0.5))): 6.586284e+002
Size of byte sequence:  301
Randomness degree:      -6.494423e+001
Randomness probability: 6.238476e-029
Conclusion: The byte sequence does not appear to be random.

C:\> CryptoID.exe test.tst 10750 10938
CryptoID v1.0 - (c) 2008 Omar A. Herrera Reyna (oherrera@prodigy.net.mx)
Median(Modulus(FrFt(Sequence,0.5))): 8.563712e+002
Size of byte sequence:  189
Randomness degree:      5.630021e+001
Randomness probability: 1.000000e+000
Encryption degree:      5.817181e+003
Encryption probability: 1.000000e+000
Conclusion: The byte sequence appears to be truly random (i.e. it could be data
encrypted with a robust cryptographic algorithm).
```

Also, note that probabilities here are not treated the same as in CryptoLocator. Please look at the source code and [HER 06] for more information on this technique.

One note about [HER 06]: The material as far as I know is not publicly available. Even though I'm the author, being a Dissertation gives the University of Bradford certain rights on the research it describes. Please contact the course tutor for the MSc in Forensic Computing at the University of Bradford for more information if you want access to this material.

Comments and suggestions

If you would like to make comments or suggestions regarding this toolset please feel free to write me some lines at oherrera@prodigy.net.mx. It is possible that I may not reply or take a long time to do so because of my workload, but I make my best to read each email I receive.

Alternatively, if you know Spanish, you could visit my Blog (http://candadodigital.blogspot.com) and leave me a message there as well.

References

[HER 06] Herrera, O. (2006). Forensic Analysis Techniques for Cryptographically Armoured Malicious Software. 15 Sep 2006. Dissertation submitted in partial fulfillment of the requirements for the degree of Master by Advanced Study in Forensic Computing. University of Bradford, United Kingdom.

[RUK 01] Rukhin, A. et. al. (2001). A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications, NIST Special Publication 800-22. 15 May 2001. National Institute of Standards and Technology.