PRINCIPLES AND PRACTISE OF X-RAYING

Frédéric Perriot, Peter Ferrie
Symantec Security Response, 2500 Broadway
Suite 200, Santa Monica, CA 90404, USA

Email fperriot@symantec.com,
pferrie@symantec.com

ABSTRACT

X-raying designates a virus detection method relying on a known-plaintext attack on the virus body. Far from being a new technique, x-raying has been used since the DOS days of yore to detect encrypted or polymorphic viruses without having to emulate their decryption code. As Entry-Point Obscuring viruses surfaced, another advantage of x-raying became obvious, namely the ability to detect an infection without the – sometimes prohibitive – cost of locating the decryption code in the infected object.

In this paper we examine conventional approaches to x-raying and present our own improvements and additions to the traditional methods. We also describe precise applications of x-raying to the detection of several recent polymorphic Win32 viruses. Finally we discuss the potential and limits of x-raying when faced with complex polymorphic viruses employing multiple encryption layers or metamorphism.

INTRODUCTION

No. of Win32 virus variants as of 1 June, 2004	almost 10^4
No. of cells in a human being	10^14
No. of atoms in planet Earth	10^50
No. of atoms in the Sun	10^57
No. of atoms in the Milky Way Galaxy	10^69
No. of atoms in the Universe	10^78
No. of possible W32/Efish.A encryptions	10^507

Conventions

Numbers in the text are in decimal, unless they have a 0x prefix, in which case they are in hexadecimal. Numbers in illustrations are in hexadecimal. Unless otherwise noted, P designates the plaintext, C the ciphertext, and K the key. p0...pn designate individual symbols of the plaintext, c0...cn symbols of the ciphertext.

PRINCIPLES OF X-RAYING

What is x-raying?

Intuitively, the term ‘x-raying’ applied to computer virus detection designates an operation analogous to the use of x-rays in medical science. In physics x-rays are high-frequency electromagnetic radiations. By bombarding a human body with x-rays, and measuring the ratio of x-rays that goes through, it is possible to obtain a picture of bones, teeth, or internal organs, seeing through the skin and outer tissues. This is due to the fact that various parts of the body absorb more or less radiation.

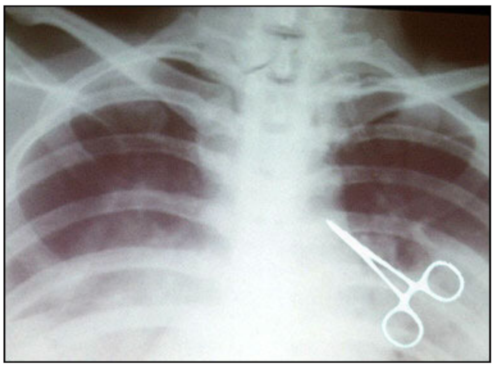


Figure 1. Occasionally, medical x-rays pick up intruders too.

Likewise, in the detection of self-encrypted computer viruses, x-raying is a set of techniques that provide a ‘picture’ of the virus body, seeing through the layer, or layers, of encryption. X-raying is applicable if the encryption algorithm used by a virus presents certain weaknesses. The object of this paper is to expose the common principles of x-raying, as used in computer virology, through practical examples that the authors applied to virus detection in the past.

A simple x-ray example

The simplest form of encryption that is amenable to x-raying is a byte ‘exclusive or’ (XOR). In this encryption method, each byte of the encrypted text (‘cyphertext’) is derived from one byte of the clear text (‘plaintext’) by XOR’ing it with a fixed byte value between 0 and 255 (the ‘key’). This method is used in countless viruses, even today.

As an example, consider the plaintext P:

P=	e8	00	00	5d
	p0	p1	p2	p3

Figure 2. Sample plaintext.

and let us choose the key to be 0x99. Then the ciphertext C is:

C=	e8^99	00^99	00^99	5d^99
	c0	c1	c2	c3
=	71	99	99	c4

Figure 3. Sample ciphertext.

Suppose now that we are given this ciphertext C and asked whether it could be an encryption of the plaintext P with a byte XOR, with any key. We can determine this in two steps, by guessing what the value of the key k must be according to the value of first byte of ciphertext c0. On the assumption that c0 = 0xe8 ^ k, we guess that k = c0 ^ 0xe8 = 0x71 ^ 0xe8 = 0x99. Then we verify that the rest of the ciphertext decrypts correctly by applying the guessed key k to the following bytes:

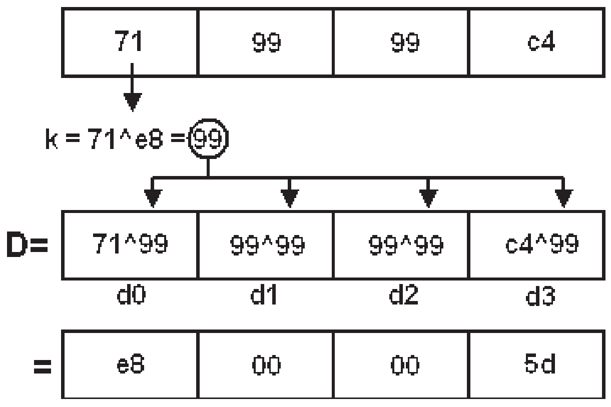


Figure 4. Sample x-raying.

The text D, which we tentatively decrypted, matches the plaintext P, hence we found the pattern in the ciphertext without knowing the encryption key *a priori*. Note that we really checked three bytes instead of the total four, because we had to guess a one-byte-long key.

A short history of x-raying

X-raying has been used since the days of encrypted DOS viruses; viruses which employed a fixed decryption loop preceding their body, and whose body was encrypted with a fixed key. At the time, x-raying was an alternative to picking a detection string from the, often short, decryption loop. Searching for a string from the longer encrypted body reduced the risk of false-positives.

Oligomorphic and polymorphic viruses then appeared, which mutated their decryption loop, and thus made the detection of the decryption loop difficult. X-raying remained possible for this class of viruses, and competed with generic decryption based on emulation, at least for viruses using simple enough encryption methods.

Variants of x-raying were used as early as 1991 to detect oligomorphic viruses such as Tequila [1], and even 1988 to repair the encrypted virus Cascade [2]. The IBMAV team has a patent on x-raying dating from 1995, describing some techniques that work against various encryption methods, and how x-raying can be used systematically to scan objects for the presence of multiple patterns, including patterns containing wildcards [3].

X-raying made it into several AV engines; *F-prot*, for instance, supports x-raying of ADD and XOR encryption with byte, word and dword sizes (8-bit, 16-bit and 32-bit.) However, Fridrik Skulason indicates that x-raying can easily be defeated by using several encryption layers [4] (we mostly agree, but in some special cases, like W95/Drill, multiple layers are still x-rayable). Reportedly, Frans Veldman and Eugene Kaspersky developed similar x-ray techniques in their scanners [2, 4].

Later on, the appearance of Entry-Point Obscuring (EPO) viruses provided one more reason to use x-raying: in some cases, the decryption loop (and with it, the decryption key) is buried so deeply inside the infected objects that parsing the objects to find the key is more costly than attacking the body of the virus with x-raying methods. Such methods are usually highly dependent on the specific virus at hand, but we will try to expose some commonalities.

The existence of viruses that produce buggy decryptors is yet another reason for using x-raying. Emulation may be impossible if the decryptor generated by the virus runs into infinite loops or crashes. X-raying is interesting in this case, particularly if it permits the decryption of data necessary to repair infected files.

X-raying techniques are periodically rediscovered [5] and adapted to fit the peculiarities of new viruses.

Relation with cryptography

To describe x-raying in a more scientific way, we may borrow some vocabulary from the field of cryptography, and classify x-raying as a 'known-plaintext attack' – that is, an attack where the clear text of an encrypted message (the virus body) is known, and the goal of the attack is to recover the encryption key.

This is somewhat misleading because the key matters little in detecting an encrypted virus, the central question is whether the virus is present in the object at all. Recovery of a valid key is usually the first step in verifying the presence of the virus: once the key is recovered, by using information from a small part of the virus body, adjacent blocks of data are tentatively decrypted using the key, and then matched against patterns from the virus body.

Another subtle difference between the cryptographer's known plaintext attack and x-raying is that a known plaintext attack operates on one given block of ciphertext, whereas x-raying typically considers many possible positions of the scanned object in succession, treating them in turn as the putative ciphertext. We call this a 'sliding x-ray'. Older DOS viruses with fixed-sized

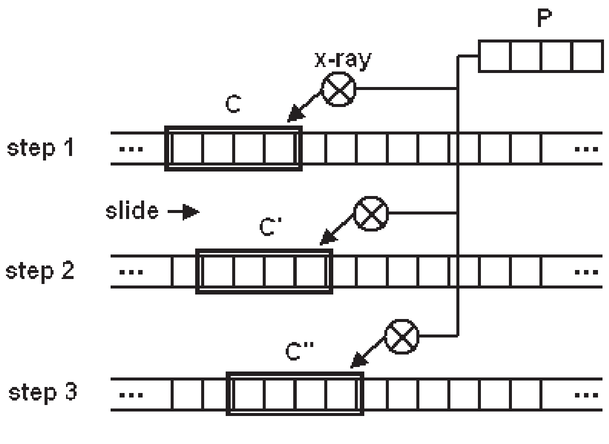


Figure 5. Sliding x-ray.

decryption loop were sometimes x-rayable at a single position, but the introduction of polymorphism, with variably sized decryption loops, made sliding x-ray a necessity.

There are a few other specificities of x-raying where the analogy with cryptography does not quite hold:

1. Cryptanalysis (the part of cryptography interested in breaking ciphers) is concerned with breaking one cipher. X-raying is concerned with breaking all possible ciphers that a virus may generate. Even early oligomorphic viruses like Tequila were able to pick one of several encryption methods to encrypt their body, like ADD or XOR. This means that the same plaintext gets encrypted using many algorithms and keys, and x-raying has to break all of them.
2. The more varied the encryption algorithms of a virus, the more difficult it is to x-ray. This goes against the cryptography principle that all the security should reside in the key, not in the algorithm [6]. In the case of complex polymorphic viruses employing running keys, the randomization of the encryption algorithm is much harder to break than the key *per se*. Alternatively, one may consider that the encryption key is split into two parts: some bits defining an algorithm (‘code key’), some bits defining parameters to the algorithm (‘data key’, the usual ‘key’ of cryptography).

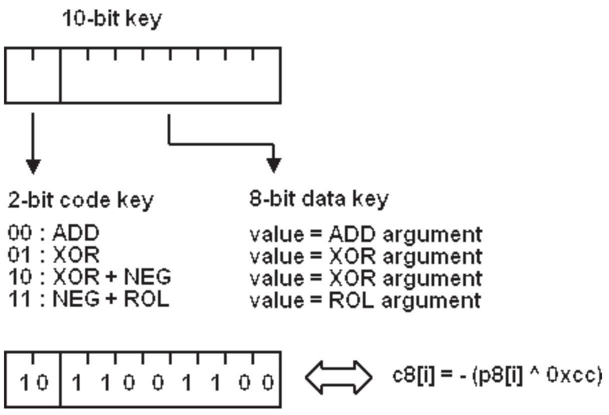


Figure 6. Code key and data key.

3. In cryptography, an algorithm must perform consistently across almost all keys and messages to be considered strong. If half of the key-space leads to a trivially breakable encryption, the algorithm is bad. In encryption algorithms produced by viruses, it is enough that some keys provide good encryption, even if the others are trivially breakable, because the goal of x-raying is to detect 100 per cent of the samples of a virus, and anything less is a false-negative.
4. Cryptographic ciphers are designed with the following goals in mind [6]: the algorithm should be fast, and the key should be small. That is, as fast and as small as possible without compromising the security of the cipher. Viruses do not share these requirements: the encryption algorithms produced by complex polymorphic viruses can be as slow and awkwardly implemented as needed, this is even an advantage from the point of view of the virus. The keys may be many hundreds of bits long, or more.

Approaches to x-raying

There is more than one way to perform x-raying of a given virus, as the examples in this paper testify. We propose to distinguish between three modes of x-raying: ‘key recovery’, ‘key validation’ and ‘invariant scanning’.

Key recovery is the most natural method, and the one we demonstrated on the simple example above, where we x-rayed a byte-wide XOR encryption. The first step is to guess a key using one part of the ciphertext and some knowledge of the plaintext. Then the key is applied to the rest of the ciphertext to recover the rest of the plaintext.

Key validation attempts to recover several keys, or pieces of keys, for several corresponding positions of the plaintext and the ciphertext, and then verifies that all the keys so recovered are consistent with one another, with respect to the encryption method considered. If they are consistent the ciphertext can be obtained by encrypting the plaintext, thus a match exists.

Going back to the simple XOR example, the key validation method considers corresponding bytes of the plaintext and the ciphertext, and XOR them together to derive a byte key for all text positions. Next, the validation phase compares k0, k1, k2, and k3. Since they are equal, they are consistent with the fixed byte XOR encryption. Therefore the encrypted pattern is present.

Invariant scanning is a very fruitful approach described in [3]. It consists of ‘reducing’ the ciphertext in such a way that the result of the reduction does not depend on the key used to encrypt the ciphertext. The reduced ciphertext can then be matched against the reduced plaintext. The result of the reduction is an invariant of the encryption function, hence the name of this approach.

Once again, consider the simple XOR example. Let us pick as a reduction the XOR of a text with itself shifted

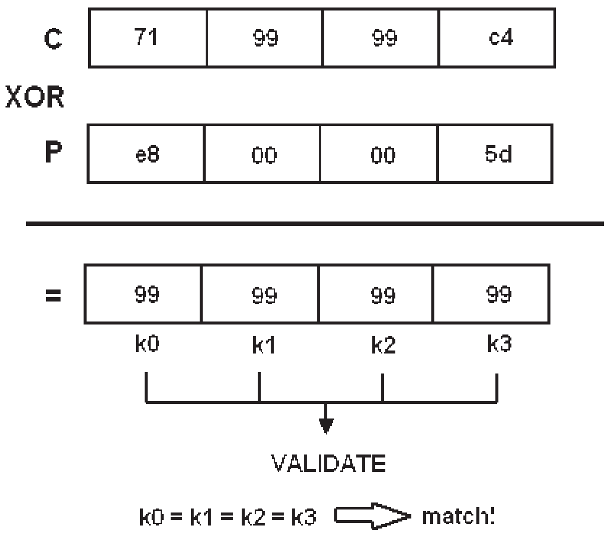


Figure 7. Sample x-ray using key validation.

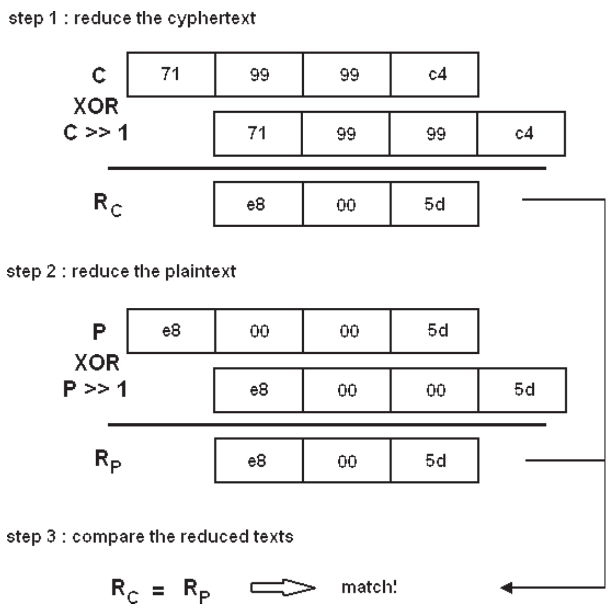

step 1 : reduce the cyphertext
C
71
99
99
c4
XOR
C >> 1
71
99
99
c4
R_C
e8
00
5d
step 2 : reduce the plaintext
P
e8
00
00
5d
XOR
P >> 1
e8
00
00
5d
R_P
e8
00
5d
step 3 : compare the reduced texts
R_C = R_P
match!

Figure 8. Sample x-ray using invariant scanning.

one byte to the right, and apply the transformation to the ciphertext and the plaintext. Then, compare the reduced ciphertext and plaintext. Since they match, we found the encrypted pattern shown in Figure 8.

For the sake of the demonstration, we included the reduction of the plaintext as one of the steps of invariant scanning. In practice the reduction of the plaintext can be done only once, when the search patterns are first compiled, and the result is then compared against several reduced ciphertexts [3].

Of the three modes, key recovery, key validation and invariant scanning, the most reliable, but also the most costly and difficult to achieve, is key recovery. Key recovery is usually necessary for the purpose of exact variant identification and repair, since it is the only way to access the fully decrypted virus body.

In cases where exact identification and repair are not necessary, key validation and invariant scanning can be comparatively easier to implement. [3] suggests a two-phase approach, where invariant scanning is performed first to take advantage of its speed, and if a possible match is found, a more thorough analysis is carried out to corroborate the match, possibly by attempting to recover the encryption key.

Applicability of x-raying

Ciphers used by viruses are, in general, weaker than modern cryptographic ciphers. Not much care is put into the design of the algorithms, and virus writers are amateur cryptographers at best. However, not all attacks that would be considered practical for cryptanalysis are usable for x-raying because of the time constraints imposed on virus scanning. X-raying competes with scanning and emulation, which run in milliseconds to seconds. Any cryptographic attack taking more than, at most, a few seconds to execute is out of the question.

As we shall demonstrate in the rest of this paper, the applicability of x-raying to the detection of a virus is often determined by the presence of mistakes or peculiarities in the encryption of the virus. Luck is a factor, and as the saying goes ‘your mileage may vary’. Nevertheless, the fact that about half of the recent complex Win32 viruses were x-rayable testifies to the usefulness of x-raying. Or maybe we were just very lucky.

In general, x-raying is not a replacement for emulation because it is too specific to virus encryption methods. Emulation is applicable to a wider range of problems. There are some classes of problems, though, to which emulation is not well suited. Consider the class of viruses using Random Decoding Algorithms (RDA) [7]. These viruses carry out a brute-force attack on their own body to find which key was used to encrypt it. Emulating the brute-force attack is very expensive, sometimes impossible, and x-raying is a better option, if at all possible.

PRACTICAL X-RAYING

Importance of the geometry

We call ‘geometry’ the positions and sizes of segments that characterize infected objects, such as position of the virus decryptor and body, minimum and maximum sizes of the decryptor, the minimum infected file size, etc.

One point that is often overlooked while discussing x-raying is the effect that disk I/O has on the time required. Since disk I/O can be very slow, it must be reduced as much as possible. Thus is it very important to choose carefully the ranges in which an x-ray should be applied. Unfortunately, the ranges are also very virus-dependent: some viruses will appear at the end of the code section, some viruses appear at the start of the data section, some viruses appear at the end of the file, etc.

The W32/Bagif and W95/Perenast families of viruses, for example, place their decryptor at or near the end of the first section, and place the encrypted virus body at the end of the last section. While simply attempting to x-ray the end of the last section of every file is certainly a possibility, the cost would be too great. To exclude some files, one can use the characteristics of an infected file that make it stand out from a non-infected file. The checking of these characteristics is the first step in the detection process. By filtering out all definitely non-infected files, what remains is (ideally) a small subset of files, on which a bit more time can be spared.

Further filtering can be performed to avoid attempting to x-ray regions that cannot possibly contain the virus body. The most common example of this type of filtering is to calculate the ratio of zeroes to non-zero bytes. If the zero count exceeds a certain threshold, then it can be decided that the range does not contain the virus body. This idea can be extended to checking the number of all possible bytes to isolate the likely locations of encrypted data, since encrypted data will usually have a fairly random byte distribution. The use of a frequency histogram makes it very simple to check these counts very quickly.

The W32/Efish family of viruses place their virus body somewhere in the last section. The location of the body depends on the structure of the host. Specifically, if the relocation table is present in the last section, then Efish will move the relocation table down, to form a gap in which the virus will place itself. In addition to the virus body, Efish constructs a substitution cipher table (see below) which is placed randomly either before or after the virus body. Efish places random data between the virus body and the table, and on either side of the entire block.

Another kind of filtering becomes useful here: the data directories in the Portable Executable (PE) header include the size of the data. Two of those tables (resources and relocations) require that the size field be correct, and coincidentally the data often appear at the end of files, when they are present. If either of these tables points into the last section, then their size can be used as a starting point (or ending point, in the case of the relocations table in an Efish file) for the scan [8]. Several other viruses, including W95/MTX, W32/Simile, and W95/Perenast, are vulnerable to this kind of shortcut. Once again, the ratio of zero bytes to non-zero bytes can be used to determine if viral data are possibly present in that region.

The W95/Perenast family of viruses fill the end of the last section with random data, before partially overwriting them with the virus body at a random position. This is similar to Efish, in the sense that there is possibly random data before and/or after the virus body, which leads to another point: it is not necessary to x-ray an entire potential region for a single pattern, chosen, for example, from either the start or the end of the virus body. Instead, a pattern can be chosen from each of the start and the end (and even some from the middle, if the body is large enough). One can then search for these patterns in parallel, starting from somewhere near to the middle of the x-ray region. If the virus has placed itself near to the end of the region, then the start pattern will be hit; if the virus has placed itself near to the start of the region, then the end pattern will be hit; if the virus is somewhere in between, then a middle pattern will be hit. This is a way to trade I/O for CPU time.

An x-ray example: W95/Perenast

The W95/Perenast family of viruses use a complex polymorphic decryptor, and the variants prior to Perenast.25239 implement a very simple decryption algorithm. This weakness in the implementation allows the virus body to be x-rayed without reference to the decryptor.

The variants prior to Perenast.25239 encrypt each dword of the virus body by XORing it with a key. On each round, the newly encrypted value is added to the current key to produce the next key. Perenast.15724 and later variants follow this with a rotation of the key. Perenast.15724/15879/16224/16254/23317 use a value of 1 as the argument of the rotation; Perenast.25026 uses a value of 2.

An x-ray routine guesses the initial value of the key by XORing the first two dwords of ciphertext with the first two dwords of plaintext, and computing the difference. Then it can proceed to decrypt the rest of the ciphertext, checking on each step that the plaintext matches the expected search pattern for the virus body.

Attacking the weakest layer: W32/Bagif

In viruses employing multiple layers of encryption, each layer usually encrypts both the decryptor of the next layer and the body. Occasionally, though, the author of a virus overlooks the fact that encrypting the body several times can make it harder to x-ray. Such is the case for W32/Bagif: a careful examination reveals a chink in its armour.

The W32/Bagif family of viruses use two layers of encryption. The first layer is a polymorphic decryptor that builds a second layer decryptor. The second layer decryptor decrypts the virus body. The first layer decryptor is extremely complex and is not subject to x-raying, however the second layer is very simple, and the algorithm is constant. This weakness in the implementation allows the virus body to be x-rayed without reference to the decryptor.

The second layer decryptor uses a so-called 'running key' encryption, where each byte of the body is XORed with a key that changes over time. The key is 32 bits long, and on every round, the low eight bits are used as an argument to the XOR operation. After each round, the key is rotated right by one bit, then a counter is subtracted from it. The counter decreases from the size of the virus to zero.

From the last byte of the virus body, the final eight-bit XOR key can be recovered immediately. By x-raying backwards, increasing the counter, and keeping track of known key bits, one can validate one extra bit of information on each round. Each pair of corresponding plaintext and ciphertext bytes yields one more bit of key, because of the key rotation. Seven bits of the combined plaintext and ciphertext are correlated with the currently known key bits. If they are not compatible, the search pattern cannot be present at the starting x-ray position.

After 24 bytes have been correlated, the entire 32-bit key is available, allowing full decryption of the virus body, if required. The pseudo-code for a possible x-ray routine for Bagif looks something like this:

```
m = 0x8000007f      (mask of known key bits)
k = p[i] ^ c[j]     (recover first eight bits of running
key, and i begins at 2, since the first byte is known)
for each byte in the plaintext pattern
k = k ror 1
k = (k & ~0xff) + ((k & 0xff) - i)
i = i + 1           (adjust counter)
j = j - 1           (going backwards in the buffer)
k' = p[i] ^ c[j]    (combine plaintext and ciphertext to
obtain current XOR key)
if (k' & m) != (k & m)
```

`break and slide` (some XOR key bits do not match the expected running key bits)

`k = (k & ~0xff) + k'` (update running key bits)

`m = m | 0xff` (gain one more bit for the mask)

```
m = m ror 1
end for
```

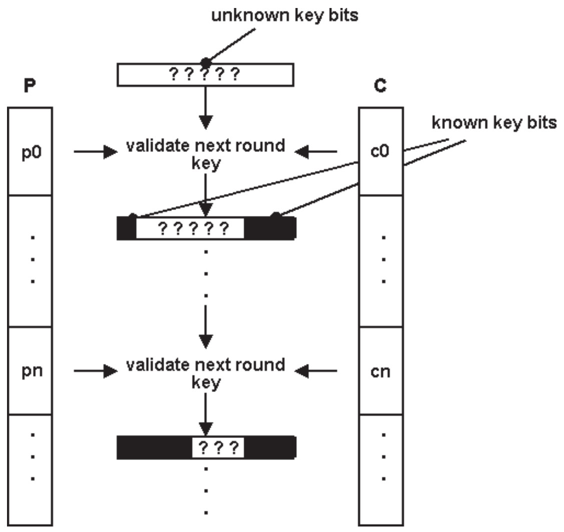


Figure 9. X-raying of W32/Bagif.

An example would proceed in this way:

`k = 0x000000d8 ^ 0xb6 = 0x0000006e` (recover first eight bits of running key)

First pass:

```
k = 0x0000006e ror 1 = 0x00000037
k = 0x00000000 + 37 - 2 = 0x00000035
i = 2 + 1 = 3
k' = 0xeb ^ 0x5e = 0xb5
diff = 0xb5 ^ 0x35 = 0x80
now 0x80 & 0x7f = 0x00, so continue
k = 0x00000000 + 0xb5 = 0x000000b5
m = 0x8000007f | 0xff = 0x800000ff
m = 0x800000ff ror 1 = 0xc000007f
```

Second pass:

```
k = 0x000000b5 ror 1 = 0x8000005a
k = 0x80000000 + 0x5a - 3 = 0x80000057
i = 3 + 1 = 4
k' = 0xff ^ 0x28 = 0xd7
diff = 0xd7 ^ 0x57 = 0x80
now 0x80 & 0x7f = 0x00, so continue
k = 0x80000000 + 0xd7 = 0x800000d7
m = 0xc000007f | 0xff = 0xc00000ff
m = 0xc00000ff ror 1 = 0xe000007f
... :
k = 0xb6e00041 ror 1 = 0xdb700020
k = 0xdb700000 + 0x20 - 0x0e = 0xdb700012
i = 14 + 1 = 15
k' = 0x00 ^ 0x92 = 0x92
diff = 0x92 ^ 0x12 = 0x80
now 0x80 & 0x7f = 0x00, so continue
k = 0xdb700000 + 0x92 = 0xdb700067
m = 0xfff8007f | 0xff = 0xfff800ff
m = 0xfff800ff ror 1 = 0xfffc007f
```

```
... :
k = 0xd59b6e10 ror 1 = 0x6acdb708
k = 0x6acdb700 + 0x08 - 0x1a = 0x6acdb7ee
i = 26 + 1 = 27
k' = 0xdb ^ 0x35 = 0xee
diff = 0xee ^ 0xee = 0x00
now 0x00 & 0xff = 0x00, so continue
k = 0x6acdb700 + 0xee = 0x6acdb7ee
m = 0xffffffff | 0xff = 0xffffffff
m = 0xffffffff ror 1 = 0xffffffff
```

and so on.

Substitution ciphers

A substitution cipher works by associating to each symbol in the plaintext alphabet one unique ciphertext symbol. The encryption consists of replacing each plaintext symbol with its substitute. You can think of it as a series of table lookups where the values in the table replace the indexes. The table is the encryption key. For instance, using 'A', 'B', 'C', 'D', 'E', 'F' as our alphabet, if we want to encrypt the message 'DEADBEEF' with the key 'BDFACE', we proceed as depicted below towards the result 'ACBADCCE'.

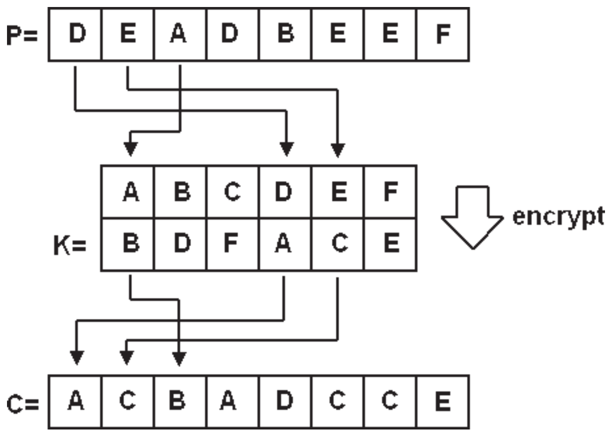


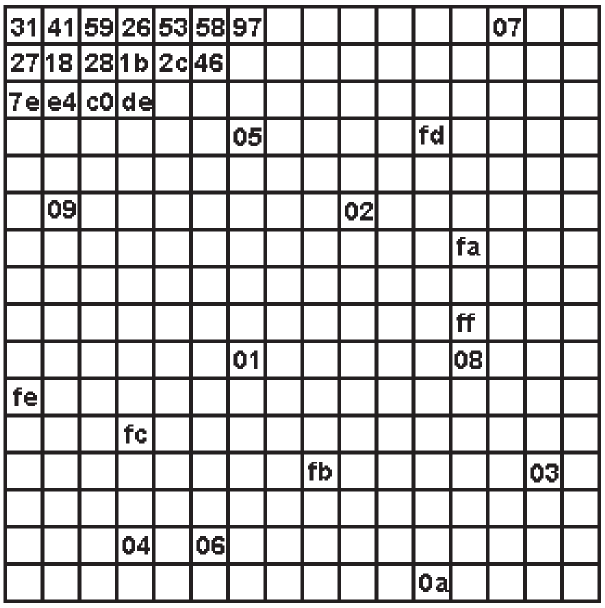
Figure 10. Example of a simple substitution cipher.

The decryption works in a similar way, with the inverse key 'DAEBFC'.

The W32/Efish.A virus, from the W32/Chiton family, uses a substitution cipher as its encryption method. To be precise, Efish encrypts its body byte-by-byte, using a 256-byte substitution table as the key. The table implements a random substitution: each byte value appears once and only once in the table, in a random place. Figure 11 is a sample table represented in two dimensions (not all values shown).

The W95/Fono virus used a substitution cipher before Efish [2]. However, in Fono, the choice was limited to a few substitutions which are their own inverse, so that the same table could be used for encryption and decryption.

The use of a byte substitution table makes the encryption much stronger than, say, a simple XOR. One way to see this is to compare the key size of a typical dword XOR (32 bits) to the equivalent size of a substitution key. Since there are 256! (factorial of 256) possible substitution



31	41	59	26	53	58	97							07		
27	18	28	1b	2c	46										
7e	e4	c0	de												
						05					fd				
	09								02						
												fa			
												ff			
						01						08			
fe															
			fc												
								fb						03	
			04		06										
											0a				

Figure 11. Sample byte substitution table of W32/Efish.

tables, approximately log2(256!) bits are needed to represent all possible values of the key. This is about 1684 bits. In other words, there are a huge number of possible ciphertexts that an x-ray routine must detect.

However, in the case of Efish.A, there is an elegant alternative to detecting all possible ciphertexts: detecting the key. Studying the geometry of the virus infection method, we see that the substitution table is stored as a contiguous array of bytes, and embedded in infected hosts close to the encrypted virus body. In fact the same region, located in the last section of the host file, must be x-rayed, whether we look for the key or the ciphertext. Here is what an infected host could look like:

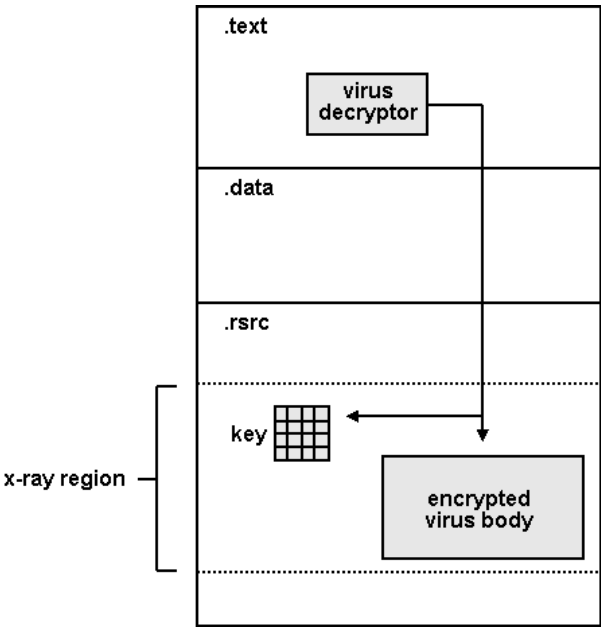


Figure 12. Geometry of file infected with W32/Efish.

To locate the key we use a sliding x-ray on the x-ray region. At every position we examine the data to determine if it can be a potential key. We do this by looking for duplicate byte values within 256 bytes of the potential key position. If there are duplicates, the data cannot be a substitution table, and we proceed to the next position. If there are no duplicates, we possibly found a key, so we use it to decrypt the x-ray region and scan for patterns of the virus body in the decrypted data (or better yet, to encrypt the search patterns and look for them in the x-ray region, which is both faster and less disruptive than decrypting the region).

Looking for duplicate byte values in an array of 256 bytes at every slide position is the tricky part. The routine that does this should be fast in order for the x-ray to be efficient. This is especially true if this routine is implemented in an interpreted language, rather than compiled to native code. A naïve algorithm to perform the duplicate bytes lookup in array A could be (in pseudo-code):

```
seen_byte[0..255] = {false}
for (i = 0; i < 256; ++i)
        if (seen_byte[A[i]])
                found duplicate, break and slide to
                                        next position
        else
        seen_byte[A[i]] = true
end for
no duplicate, found potential key in array A
```

Repeating this routine at every slide position, we get a working detection for the key. Unfortunately, it is not very efficient because we end up looking at most bytes many times, from different slide positions. Consider the following data to scan:

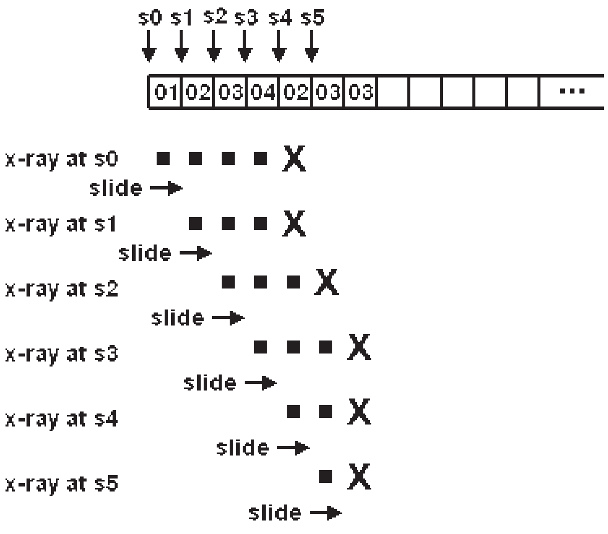


Figure 13. Substitution key search – naïve algorithm.

At slide position s0, it takes five bytes to find the duplicate byte 0x02. At slide position s1, it takes four bytes, and it is in fact the same pair of duplicate bytes with value 0x02 that stops the scan. At position s2, it takes four bytes to find duplicate byte 0x03, etc.

If we are scanning random data, the probability of finding a duplicate pair after n bytes is 1 – (1 x 255/256 x 254/256 x (256–n+1)/256). It becomes greater than 0.5 after 20 bytes, which indicates that we need to look at about 20 bytes for every slide position. In other words, we need to look at each byte approximately 20 times. (This is an approximation, due to the relationship between two consecutive slides, as can be observed in the example above.)

Wouldn't it be nice if we could make a better use of the duplicate pairs we find to slide ahead faster? When we find the duplicate pair of 0x02s at slide position s0, there is really no need to try slide position s1, because the duplicate pair is part of the 256 bytes range scanned at position s1 too.

Better yet, if we scan the array *backwards*, we can take advantage of duplicate pairs appearing late in the array to rule out many more slide positions. This is the same idea as the 'bad character heuristic' of the Boyer-Moore string scanning algorithm [9].

The optimized algorithm to scan array A reads:

```
seen_byte[0..255] = {false}
for (i = 255; i >= 0; —i)
        if (seen_byte[A[i]])
                found duplicate, break and slide
                                ahead by i+1 bytes
        else
        seen_byte[A[i]] = true
end for
no duplicate, found potential key in array A
```

Figure 14 shows an example illustrating the optimized algorithm.

Using this faster version of the duplicate byte lookup, we avoid looking at some bytes altogether. By the same argument as above, we find a duplicate pair after about 20 bytes, starting from the end of the buffer corresponding to the current slide position. Then we slide ahead by about 256 – 20 = 236 bytes on average. Thus we look at about 20/236 on average per byte of scanned data. Compare this to the 20 bytes per byte of scanned data of the naive algorithm.

This suggests that the optimized algorithm is more than two orders of magnitude faster than the naïve one. In practise, on Efish samples, the algorithm performs quite well.

Considerations on the choice and length of signatures

The choice of signatures in an x-ray routine depends on the kind of encryption being broken. There is some loss of information associated with guessing a key that decrypts the ciphertext. The length of the search patterns should compensate this loss of information to avoid possible false-positives.

The right length for x-ray signatures is similar to the 'unicity distance' of information theory, which specifies that, for the solution of the cryptanalysis to be trusted, the amount of broken ciphertext should equal at least the information in the plaintext plus the entropy of the cryptosystem [6]. The entropy of the cryptosystem is defined as the logarithm in base 2 of the size of the keyspace, that is a measure of the number of possible keys. For instance it is eight bits for a byte XOR.

In general, when x-raying a virus with a maximum key length of n bits, it is advisable to augment the normal search patterns by n bits. For instance, if the virus uses a combination of dword XOR (32-bit key), dword ADD (32-bit key), and ROL (8-bit key), it is a good idea to add nine bytes to the plaintext signatures.

Since x-raying is designed to work through encryption layers, the risk of an x-ray routine detecting itself is somewhat higher than for regular signatures. Don't x-ray yourself in the foot, organize your x-ray search patterns so that they won't be susceptible to false-positives – for instance by storing them in several out-of-order pieces, or encrypting them with a method that is not broken by your x-ray.

Note that care must be taken when choosing patterns in bodies that are decrypted using units larger than a byte. For some variants of a virus, the patterns can become misaligned with respect to the unit itself (e.g. dword decryption of bytes that are not on a dword boundary). This is fine for operations such as XOR, which are translationally invariant (where the alignment of the operation does not affect the result), but for operations such as SUB, the carry can affect the result.

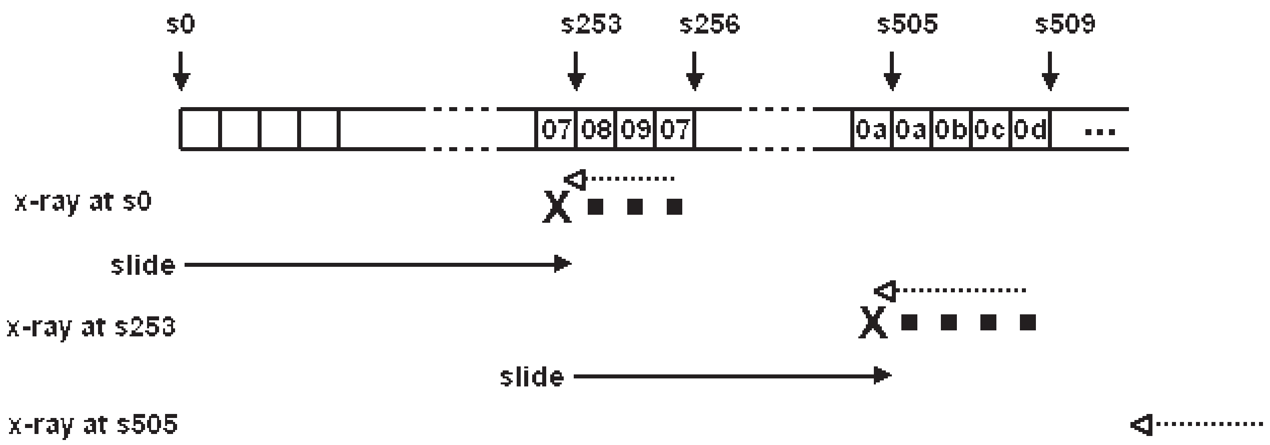


Figure 14. Substitution key search – optimized algorithm.

ADVANCED X-RAYING

X-raying with wildcards

The W32/Bagif family has several members sharing very similar search patterns. A nice generalization of the x-ray routine we presented above successfully detects variants by allowing the use of fixed-length wildcards in the search patterns. On each round of running key validation and updating, it is possible to leave out the updating phase for positions corresponding to wildcards, and just proceed with the correlation of the known key bits with the current position XOR key. Thus we gain information on the running key more slowly, but eventually the same information on the ciphertext can be checked, simply by using longer signatures. (We do not present the pseudo-code for this modified x-ray, for lack of space. The astute reader will enjoy writing it for himself. “J’ai trouvé une merveilleuse démonstration de cette proposition, mais je ne peux l’écrire dans cette marge car elle est trop longue.”)

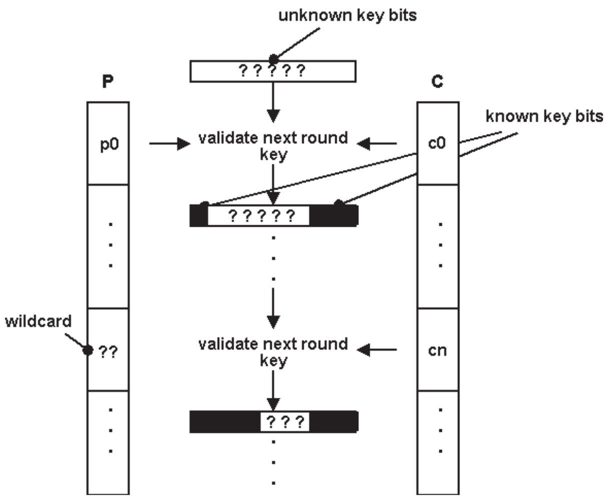


Figure 15. X-raying of W32/Bagif in the presence of wildcards.

Using specificities of the plaintext

As we mentioned earlier, invariant scanning can be used as the first step of an x-ray routine to filter out uninteresting regions. One type of invariant scanning deserves special mention: the search for patterns of ciphertext that do not depend on the encryption key. These patterns are shared by all replicants of a virus, and since most scanners have very fast string-searching primitives, looking for them is easy and fast.

It is puzzling that such patterns exist, given that encryption and polymorphism were invented precisely so that no two samples of the same virus look alike. They usually occur as a result of a weak encryption method, in conjunction with specificities of some spots in the plaintext, like series of zeroes or repeated patterns.

As an example, consider the simple encryption used by earlier variants of W95/Perenast. Each dword of ciphertext is obtained by XORing a dword of plaintext with a running key. The resulting dword of ciphertext is then subtracted from the key. The pseudo-code for the encryption reads:

```
k = initial random value
for each dword of plaintext p
        c = p ^ k
        k = k - c
end for
```

One weakness of this algorithm is that a zero dword of plaintext will result in the running key becoming zero. If p = 0, c becomes k, and the running for the next round is k - k = 0. After this happens, all the ciphertext is key-independent.

Even if the body of the virus does not contain a whole zero dword, some bits of the key will cancel themselves for well-chosen spots of the plaintext. For instance, if the low n bits of p are zero, the low n bits of the next round key will be zero, leading to a fixed n-bit pattern of ciphertext common among all virus samples.

Running keys

A common type of encryption used by polymorphic viruses relies on so-called ‘running keys’, analogous to the ‘stream ciphers’ of cryptography. Essentially, a Pseudo-Random Number Generator is used to provide a series of values, which are then combined with the plaintext on each step of the encryption process to produce the ciphertext.

While cryptographic stream ciphers use well-known primitives such as Linear Feedback Shift Registers (LFSR) (see [6]), virus authors do not embarrass themselves with theoretical considerations, and just use any random combination of arithmetical operations to modify the running key: XOR, ADD, rotations, MUL, DIV, shifts, bit swapping, and possibly other assembly instructions.

Depending on how many operations are usable to modify the running key, it may be possible to x-ray the virus by enumerating all combinations of operations, and trying them all in turn. We call this method ‘brute-forcing the code key’ because it involves a systematic enumeration of the key bits associated with the choice of encryption operations, which we called the ‘code key’ earlier.

The data key usually does not have to be brute-forced: for a given set of operations, the values of the parameters of the operations can be derived by combining the ciphertext with the plaintext.

While the recovery of keys from multiple operations is fairly straightforward, the size of the search pattern needs to be sufficiently large to ensure that the recovered keys are, in fact, correct. Let us examine an example from W32/Magistr. The encrypted bytes and the plaintext bytes are shown in Figure 16.

We know that this virus can produce dword XOR decryptors involving ROL and ADD for the running key modification, i.e.

Encrypted bytes

```
E4 41 DB 19 49 68 62 E4 DA E7 25 B9 3A 10 15 C3 57 9E 29 3B EA 59 20
01 DE E2 51 2E 66 80 31 D4 50 B2 78 8A 30 4B 51 06 4B 69 B1 19 C3 CC
A1 CE 54 44 22 56 0E A6 55 0C 2C DE 5C D4 AA AA CB 22 7E A5 79 C4 7D
AF B8 E4 C5 92 C7 ED 79 23 2F 01 71 B9 AF 00 BE 39 2E C1 A0 CB AC C8
E6 13 0B F9 AF BE D2 C4 87 D6 34 4F 5A 2D D0 89 3E 80 95
```

Plaintext bytes

```
E8 38 55 00 00 E8 B3 53 00 00 85 C0 74 08 90 90 90 90 8B 64 24 08 60
9C E8 C3 53 00 00 85 ED 75 0D 90 90 90 90 8B 74 24 24 EB 07 90 90 90
8B 74 24 2C 81 E6 00 00 FF FF FC E8 4E 17 00 00 89 85 C8 86 40 00 89
85 C4 86 40 00 8C C9 81 E1 00 FF 00 00 E3 0A C7 85 C4 86 40 00 00 00
F7 BF 8D B5 A8 73 40 00 8D BD E4 83 40 00 81 BD C4 86 40
```

Figure 16. An example from W32/Magistr.

```
for i = 0 to virus size
          p[i] = c[i] ^ k1
          k1 = k1 + k2    (these two lines
          k1 = k1 rol k3  can be swapped)
end for
```

Without knowing the order of the ADD and the ROL, we must try both.

Let's begin by assuming that the order is ADD then ROL.

We recover the original key by taking the first encrypted dword and XOR'ing with the first dword of plaintext:

```
0x19DB41E4 ^ 0x005538E8 = 0x198E790C
```

The order of the operations, ADD then ROL, indicates that the rotation yields the second round key. Conversely, we need to apply an inverse rotation to the second round key in our x-ray.

Trying all ROL arguments from 1 to 31, we take the second encrypted dword, XOR with the second plaintext dword, thus obtaining the second key, then rotate this value, then subtract the first round key from this value. This gives us the ADD value.

```
k2 = ((c[1] ^ p[1]) rol k3) - k1
```

where

c[1] is 0xE4626849

p[1] is 0x53B3E800

k3 is 1..31

k1 is 0x198E790C

When the ADD and ROL values are correct, applying the encryption algorithm to the third dword will give the same delta.

```
delta = ((c[1] ^ p[1]) rol k3) - (c[0] ^ p[0])
compare delta to ((c[2] ^ p[2]) rol k3) - (c[1] ^ p[1])
```

Trying k3 = 3

```
((0xE4626849 ^ 0x53B3E800) rol 3) - (0x19DB41E4 ^
0x005538E8) = 0xA4FD8941
((0xB925E7DA ^ 0xC0850000 rol 3) - (0xE4626849 ^
0x53B3E800) = 0x1535BE8A != 0xA4FD8941
```

Clearly, k3 = 3 is incorrect. However, trying k3 = 18

```
((0xE4626849 ^ 53B3E800) rol 18) - (0x19DB41E4 ^
0x005538E8) = 0xE798663A
((0xB925E7DA ^ 0xC0850000) rol 18) - (0xE4626849 ^
0x53B3E800) = 0xE798663A
```

A match! So our algorithm seems to be:

```
k1 = (k1 + 0xE798663A) rol 181
```

Let us decrypt some bytes and compare (see Figure 17). All of the bytes between brackets are incorrect. How did it happen? It's the order of ADD and ROL that creates keys that match until an overflow occurs in the ADD.

Let us try ROL then ADD instead. This time, the rotation operates on the first-round key. Therefore, our x-ray works by applying an inverse rotation to the first round key. We enumerate the possible ROL arguments, and the corresponding rotated first round keys, then subtract them from the second round key. We reverse the order of the algorithm above.

```
delta = (c[1] ^ p[1]) - ((c[0] ^ p[0]) rol k3)
compare delta to (c[2] ^ p[2]) - ((c[1] ^ p[1]) rol k3)
```

Trying k3 = 3

```
(0xE4626849 ^ 0x53B3E800) - ((0x19DB41E4 ^
0x005538E8) rol 3) = 0xEB5DB7E9
(0xB925E7DA ^ 0xC0850000) - ((0xE4626849 ^
0x53B3E800) rol 3) = 0xBB14E58D != 0xEB5DB7E9
```

Clearly, k3 = 3 is incorrect. However, trying k3 = 14

```
((0xE4626849 ^ 0x53B3E800) rol 14) - (0x19DB41E4 ^
0x005538E8) = 0x198E79E6
((0xB925E7DA ^ 0xC0850000) rol 14) - (0xE4626849 ^
0x53B3E800) = 0x198E79E6
```

A match! So our algorithm is:

```
k1 = (k1 rol 14) + 0x198E79E6
```

And applying it to the entire ciphertext yields the exact virus body.

```
E8 38 55 00 00 E8 B3 53 00 00 85 C0 74 08 90 90 90 90 8B 64 24 08 60
9C E8 C3 53 00 00 85 ED 75 0D 90 90 90 90 8B 74 24 24 EB 07 90 90 90
8B 74 24 2C 81 E6 00 00 FF FF FC E8 4E 17 00 00 89 85 C8 86 40 00 89
85 C4 86 40 00 8C C9 81 E1 00 FF[03]00 E3 0A C7[05 C5]86 40 00 00 20
F7[A7]8D B5 A8 73 40[02 0D]BD E4 83 40 00[A1]BD[3C 87]40
```

Figure 17.

More substitution ciphers

After using a substitution cipher as the encryption of W32/Efish.A, the virus author wanted to try something more complex. In Efish.C, [s]he switched from a simple substitution cipher, where each symbol is replaced by another unique symbol, to a 'homophonic substitution cipher', where a symbol of plaintext may be represented by more than one symbol of ciphertext.

The advantage of using a homophonic substitution cipher is that the frequencies of symbols in the ciphertext are no longer the same as they are in the plaintext. Two occurrences of the same symbol in the plaintext may become two different symbols in the ciphertext. Another consequence is that the decryption key of the homophonic substitution cipher no longer has to contain each symbol only once. Symbols may be repeated in the key, to map distinct ciphertext symbols to the same plaintext symbol during decryption.

Going back to the 'DEADBEEF' example, we notice that the 'C' symbol does not appear in the plaintext. Therefore, we may 're-use' the corresponding ciphertext symbol 'F' as the image of, say, symbol 'E' in the encryption key of a homophonic substitution cipher:

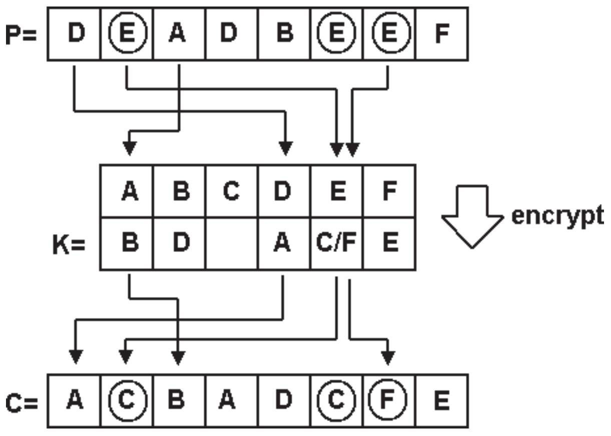


Figure 18. Encryption process of a homophonic substitution cipher.

Notice how the three 'E's in the plaintext are translated to different ciphertext symbols.

The decryption is a simple substitution. Notice the duplicate value in the decryption key shown in Figure 19.

In practice, for the purpose of detecting Efish.C, the use of a homophonic cipher defeats the algorithm described earlier, based on looking for the key as a table containing all byte values from 0 to 255 once and only once. Fortunately, it is also possible to x-ray the ciphertext rather than the key, albeit at a cost in performance.

The algorithm to x-ray all possible substituted ciphertexts works in key recovery mode. We want to determine if ciphertext C is a substitution of plaintext P. We work our way through C and P, building the potential decryption key as we go along. Since pi must be obtained from ci, we populate the decryption key index ci with the value pi.

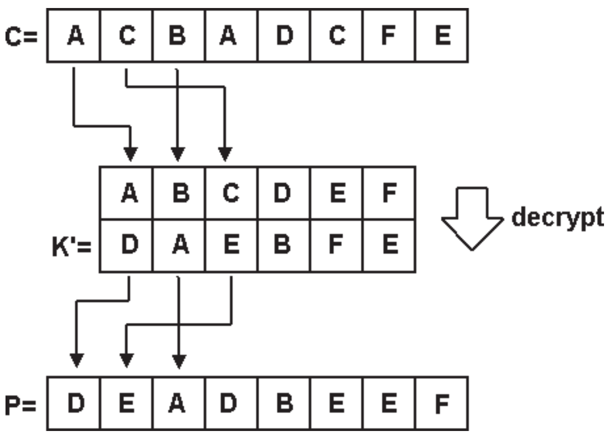


Figure 19. Encryption process of a homophonic substitution cipher.

If we hit an index that was already used, and find out that two distinct plaintext values compete for this index, the ciphertext cannot be a substitution of the plaintext (simple or homophonic), since any given ciphertext symbol must always decrypt to the same plaintext symbol.

Once we finish building the key, we proved that the ciphertext is a substitution of the plaintext, by construction of the decryption key. We can check the decryption key for duplicate plaintext values to distinguish simple from homophonic ciphers. We can also recover the full plaintext by applying the decryption key to the rest of the ciphertext (after we have worked out enough of the key to associate each ciphertext symbol to a plaintext symbol).

Applying this algorithm to a byte substitution cipher, as used by Efish, the pseudo-code reads:

```
decr_key[0..255] = {undefined}
for each pair of bytes (pi, ci) in (P, C) the
plaintext and ciphertext
   if (decr_key[ci] is defined && decr_key[ci] != pi)
      found competing plaintext symbols for the same
ciphertext symbol, exit
   else
      decr_key[ci] = pi
end for
successfully built decryption key, apply key to more
ciphertext
```

This algorithm is correct, but not very efficient, since it requires a lengthy key recovery procedure at each position of the sliding x-ray. We will see how to optimize it shortly, by making use of a weakness in the Efish.C homophonic substitution cipher.

Using the randomness, or lack thereof

Viruses often use tailor-made Pseudo-Random Number Generators (PRNG). The strength of the PRNGs varies considerably, from generators simply using clock ticks as random values, to Linear Congruential Generators, like the C runtime rand() function, to complex generators from the literature.

On one end of the spectrum (the x-ray spectrum!) Perenast uses the RDTSC instruction (ReaD Time-Stamp Counter) of Intel processors (available since the Pentium) to retrieve the processor ticks, and fills an array of values with the ticks. The array is used as a random-sized padding buffer, and gets encrypted in addition to the virus body.

Because of the short time interval between consecutive uses of RDTSC, the random buffer is in fact filled with nearby and increasing values. Statistical properties of the underlying plaintext can facilitate x-raying, although, in the case of Perenast, they are ultimately unnecessary. Had the virus been metamorphic, recourse to this trick might have saved the day.

Besides exploiting the weakness of the PRNG algorithm, it is sometimes possible to exploit a weakness in the seeding of the generator. If the initial state of the generator is one of only a few values, and if the algorithm to obtain the sequence of random numbers is deterministic, the possible outputs of the polymorphic engine (among other virus characteristics) are reduced to a few, easy to check for, values. Such is the case for W32/Marburg, which uses the current date to seed its generator.

At the other of the spectrum, Efish uses a very complex PRNG, dubbed the ‘Mersenne Twister’, after the kind of prime number at its heart. Ironically, the strength of this generator helps somewhat in detecting the virus. Here is why.

The construction of the encryption key in the Efish.C homophonic substitution cipher is intertwined with the encryption itself. The virus starts with a random substitution table. Then, for each byte to encrypt, it either uses the current table, with 94 per cent chance or, with 6 per cent chance, it searches for an unused plaintext byte in its body and reuses the corresponding ciphertext byte in the substitution table as the image of the current byte. The reused table entry is marked as such, and the ciphertext byte cannot be used again as the image of another plaintext byte.

Thus the encrypted byte associated with one clear byte can change over time, but eventually, when there are no more unused entries in the table, the key stabilizes to that of simple substitution cipher.

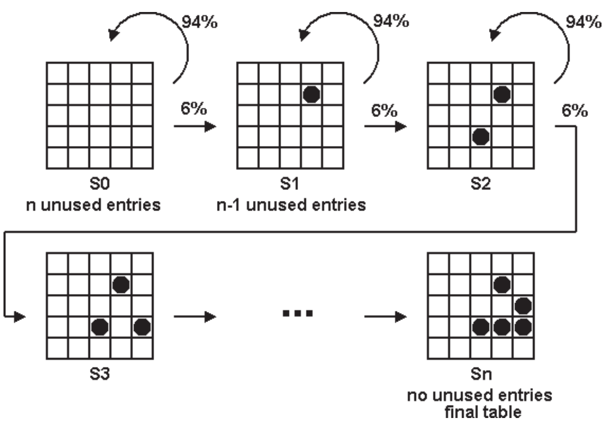


Figure 20. Decay of the substitution table of W32/Efish.C.

The virus body uses all but six byte values, and since the virus saves some bytes from the host into its body, this count of unused plaintext bytes could be even smaller. At each step, the probability is about 6 per cent that the virus will reuse any given slot in its table. Since the PRNG is very good, we can trust that this is statistically true.

Assuming the worst case of six unused bytes in the plaintext, the probability of still having an unused entry in the table after n steps of encryption is

```
0.94 ^ n + 0.06 * 0.94 ^ (n-1) + … + 0.06 ^ 5 * 0.94 ^ (n-5)
```

which becomes very small after a few hundred bytes. After 350 bytes, it is less than one in a billion. The result is that the homophonic substitution cipher decays into a simple substitution cipher after n bytes, where the value of n is small. If we pick a signature from the end of the virus, we can assume that it was encrypted using a simple substitution cipher, with the final table.

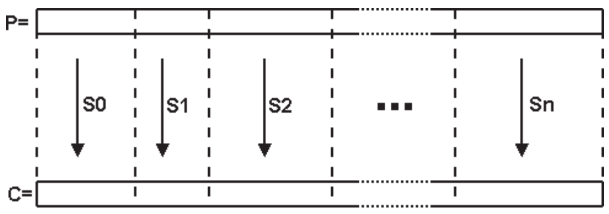


Figure 21. Simple substitution ciphers applied to segments of W32/Efish.C.

As we mentioned in the section on breaking homophonic substitution ciphers, the key recovery algorithm is expensive. Thanks to the decay in the Efish.C cipher, we can correlate the frequency of bytes, as we would do in a simple substitution cipher, before applying the key recovery algorithm. We can check, for any pair of bytes in the ciphertext, whether they are the same, or different, because a simple substitution cipher conserves the frequencies of the plaintext.

This is an example of using invariant scanning before key recovery to speed up the x-raying.

Multi-layer x-raying

Some complex polymorphic viruses, like Zhengxi or W32/Dislex, use multiple layers of encryption. These are in general impossible to x-ray in reasonable time: one would need to brute-force the number of layers, usually variable, the encryption algorithm in each layer (the code key), and possibly some parts of the data key. As in the case of running keys, this is not feasible for more than a few layers, and a few encryption operations per layer.

However, there are two special cases worth mentioning: multiple layers of simple linear encryption operations, and aligned layers. Both of these may be x-rayed if the data key in each layer is fixed (as opposed to a running key.)

In the case of simple linear operations, like XOR and rotations for instance, using constant values as an argument, multiple layers are equivalent to one. Therefore, the x-ray routine can proceed as it would in the case of a single layer with one XOR and one rotation.

In the case of aligned layers of operations using fixed keys, the cipher can be treated as a substitution cipher operating on symbols wider than a byte – usually a dword for Win32 viruses. Since a given plaintext symbol is always encrypted to the same ciphertext symbol, the frequencies of symbols are conserved in the ciphertext. It is usually possible to find repeated – and aligned – dwords in virus code, for instance in repeated assembly code snippets.

If the layers are unaligned, x-raying is much more difficult because the 'diffusion' of the overall cipher is greatly improved. In the example below, showing three layers of dword encryption, one bit of plaintext can influence the value of bits of ciphertext as far as nine bytes away.

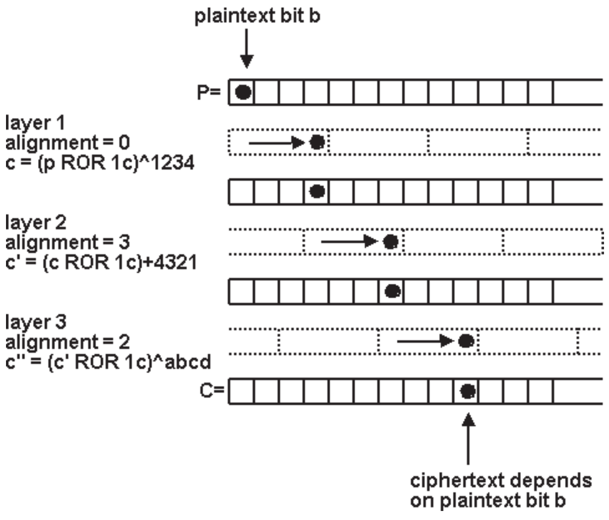


Figure 22. Diffusion effect of multiple encryption layers.

Hybrid x-raying and metamorphism

Viruses employing complex encryption methods, such as complex running key transformations or multiple layers of encryption, are in general impossible to x-ray using only the ciphertext. In some cases, though, it is possible to use information from the decryptor in an x-ray routine. Some possibilities include: recovering the code and data keys from the decryptor, recovering the code key from the decryptor to facilitate x-raying the data key (this can be done by checking for the presence of tell-tale opcodes in the decryptor).

The code key is usually easier to recover from the decryptor than the data key, because it only involves determining which atomic operations are applied to the ciphertext and/or the running key. Through instruction parsing, the operations can be recovered or guessed.

On the other hand the data key is often spread over a number of instructions, whose combined contributions participate in the data key. Without some form of emulation, and a knowledge of possible initialization values, the data key remains hidden from casual decryptor parsing.

Metamorphic viruses, which mutate their entire body, can be seen as an extra layer of confusion (opcode replacement) and diffusion (opcode swapping and routine swapping) applied to the plaintext before encryption. The mutation of the virus body shifts the problem of x-raying from a known plaintext attack to a 'partially known' plaintext attack. Traditional cryptography handles such problems by considering statistical properties of the underlying plaintext, and the repercussions of these in the ciphertext. For very simple encryption primitives, the same could be done in x-raying. Unfortunately, at the present time, we do not have an example of a metamorphic virus that would fall into this category. We look forward to receiving one in the future!

APPENDICES

W32/Bagif

W32/Bagif (see [10]) uses two layers of encryption. The first layer is a polymorphic decryptor that builds the second layer decryptor a dword at a time, using an entirely unrolled loop. The build process includes long dummy loops, and the use of transcendental functions of the floating-point unit. Some of the FPU results are inexact, making emulation slow and potentially unreliable. A sample of the first decryptor snippet is:

```
PUSH    D96651BD
POP     EBP
SUB     EBP, 16053599
MOV     EBX, A82245C5
ROR     EBX, 0D
XOR     EBX, 6AA48211
CMP     EAX, EDI
JPE     98481BC2
XOR     EAX, 31961ADB
XOR     EDX, C4C6EF8B
XOR     EDX, 4EB07668
PUSH    ECX
PUSH    EDI
XOR     ECX, EBP
PUSH    ESI
SHRD    ECX, EBX, 10
PUSH    EBP
```

The second layer is a constant decryptor and decrypts the virus body using the following algorithm:

```
p[i] = c[i] ^ (k & 0xff)
k = (k & ~0xff) + ((k + (size & 0xff)) & 0xff)
k = k rol 1
```

as can be seen here:

```
    MOV     ECX, DWORD PTR SS:[EBP + 000000CB]
    MOV     EBX, DWORD PTR SS:[EBP + 000000CF]
0012FF3D:
    LODS    BYTE PTR DS:[ESI]
    XOR     AL, BL
    ADD     BL, CL
    ROL     EBX, 01
    STOS    BYTE PTR ES:[EDI]
    LOOPD   0012FF3D
```

W95/Drill (aka W32/Tuareg, W32/Mental)

W95/Drill (see [11]) uses two layers of encryption. The first layer is a polymorphic decryptor in a long loop that includes *Windows* API calls, making emulation a difficult prospect. Its algorithm is:

```
c'[i] = c[i] op k1  (where op is ^ or +)
```

A sample decryptor snippet is:

```
ADD       CH, BYTE PTR DS:[0040FAF3]
AND       CL, DH
XOR       EBX, EBP
AND       ECX, EDX
AND       CL, 3B
XCHG      CX, SI
XOR       DWORD PTR DS:[EBX + 0040E000], EDI
ADD       DH, CL
AND       ESI, EBP
XCHG      ECX, ESI
XOR       EBX, EBP
CMP       EBP, EDX
JLE       00401F95
PUSH      0000EE5B
CALL NEAR 004020BD
```

The second layer is an oligomorphic decryptor in Drill.12292, making verification more difficult. It decrypts the virus body using the following algorithm:

```
p[j] = c'[j] ^ k2
```

A sample decryptor is:

```
   PUSH        EAX
   CALL NEAR   0040E006
   POP         EAX
   INC         EAX
   SUB         EAX, 00401007
   LEA         EBX, DWORD PTR [EAX + 00401028]
   MOV         ECX, 00000C01
   MOV         ESI, 0DC18220
0040E01E:
   XOR         DWORD PTR DS:[EBX], ESI
   SUB         EBX, -04
   LOOPD       0040E01E
```

The second layer is a polymorphic decryptor in Drill.14896/18624/20994, making verification quite difficult. It decrypts the virus body using the following algorithm:

```
p[j] = c'[j] op k2  (where op is ^ or +)
k2 = k2 op k3  (where op is ^ or +)
```

A sample decryptor is:

```
   MOV         EBX, 19DC6C42
   CMC
   PUSH        0041103C
   POP         EBP
   PUSH        000011FC
   POP         EDI
00411012:
   SUB         DWORD PTR SS:[EBP + 00], EBX
   AND         EDX, EAX
   STD
   XOR         EBX, DC5B45DB
   NOP
   LEA         EBP, DWORD PTR [EBP + 04]
   XOR         ECX, ESP
   DEC         EDI
   JNE         00411012
```

W32/Efish (aka W32/Chiton, W32/Shrug)

W32/Efish (see [12]) uses entrypoint obscuring to hide a small (< 32 bytes) decryptor, and the decryptor is oligomorphic (Efish.A) or polymorphic (Efish.B/Efish.C). It decrypts the virus body using the following algorithm:

```
p[i] = table[c[i]];
```

A sample decryptor is:

```
   PUSHAD
   STD
   MOV         ESI, 0040C0E1
   LEA         EBX, DWORD PTR [ESI + FFFFE2FA]
   ENTER       1120, 00
   MOV         EDI, EBP
004017D3:
   LODS        BYTE PTR DS:[ESI]
   XLATB
   MOV         BYTE PTR DS:[EDI], AL
   DEC         EDI
   CMP         EDI, ESP
   JNB         004017D3
   PUSH        ESP
   RETD
```

W32/Magistr

W32/Magistr.39921 (see [13]) uses a polymorphic decryptor in a long loop that includes Structured Exception Handling, making emulation a difficult prospect. Its algorithm is:

```
p[i] = c[i] ^ k
```

The transformation of k is from one to three unique operations chosen from +, ^, rot, in any order.

A sample decryptor snippet is:

```
   SHL         EAX, 71
   JMP NEAR 01018359
   XOR         DWORD PTR DS:[ECX], EDI
   SALC
   RCR         EAX, 1E
   RETD
   AND         EAX, 07557C1D
01018359:
   CMP         EAX, +24
   XOR         DWORD PTR DS:[EBX], EDI
   MOV         EAX, EBX
   ADD         EAX, 00000004
   XCHG        EBX, EAX
   CLC
   ROL         EDI, 50
```

W95/Perenast (aka W32/Stepan, W32/Stepar, W32/Stepaik, W32/Perelett)

W95/Perenast.14903/15349/15350/15383/15694 (see [14]) uses entrypoint obscuring to hide a polymorphic decryptor. Its algorithm is:

```
p[i] = c[i] ^ k
k = k - c[i]
```

A sample decryptor is:

```
0040285C:
   MOV         ESI, DWORD PTR DS:[EDI]
   XCHG        EAX, EAX
   XOR         DWORD PTR DS:[EDI], EDX
   XCHG        ESP, ESP
   SUB         EDX, ESI
   MOV         EDI, EDI
   ADD         EDI, +04
   MOV         EDI, EDI
   DEC         ECX
   JNE         0040285C
   JMP NEAR    PTR EAX
```

W95/Perenast.15724/15879/16224/16254/23317 uses entrypoint obscuring to hide a polymorphic decryptor. Its algorithm is:

```
p[i] = c[i] ^ k
k = k - c[i]
k = k ror 1
```

A sample decryptor is:

```
   MOV         EDX, F50B5638
00402503:
   MOV         ESI, DWORD PTR DS:[EDI]
   XOR         DWORD PTR DS:[EDI], EDX
   SUB         EDX, ESI
   ROR         EDX, 01
   ADD         EDI, +04
   LOOPD       00402503
   JMP NEAR    PTR EAX
```

W95/Perenast.25026 uses entrypoint obscuring to hide a polymorphic decryptor. Its algorithm is:

```
p[i] = c[i] ^ k
k = k - c[i]
k = k ror 2
```

A sample decryptor is:

```
00406538:
   MOV         ESI, DWORD PTR DS:[EDI]
   JMP SHORT   0040653F
   PUSH        SS
   OUT         DX, EAX
   PUSH        ESI
0040653F:
   MOV         ESI, ESI
   XOR         DWORD PTR DS:[EDI], EDX
   JMP SHORT   00406545
00406545:
   JMP SHORT   00406549
   CMP         DWORD PTR DS:[EDI], EBP
00406549:
   XCHG        EBX, EBX
   SUB         EDX, ESI
   MOV         ECX, ECX
   ROR         EDX, 02
   JMP SHORT   00406555
   WAIT
00406555:
   MOV         EBP, EBP
   ADD         EDI, +04
   MOV         EDX, EDX
   DEC         ECX
   JNE         00406538
   JMP NEAR    PTR EAX
```

W95/Perenast.25239 uses entrypoint obscuring to hide a polymorphic decryptor. The decryptor uses transcendental functions of the floating-point unit to calculate the index sequence, resulting in some indexes being repeated and thus some entries being encrypted more than others.

Its algorithm is:

```
p[i] = c[i] ^ k1
k1 = k1 ^ p[i]
k1 = k1 rol 3
k1 = k1 op k2 (where op is ^ or +)
(next(i) is a complex fpu transformation)
```

A sample decryptor snippet is:

```
0100BC80:
   XOR         DWORD PTR DS:[EDI*4 + ESI], EDX
   MOV         EBP, EBP
   XOR         EDX, DWORD PTR DS:[EDI*4 + ESI]
   MOV         EDX, EDX
   ROL         EDX, 03
   XCHG        ESP, ESP
   ADD         EDX, 24173F0A
   MOV         ESP, ESP
   CALL NEAR   0100BD03
   DEC         ECX
   JNE         0100BC80
```

Tequila

Tequila uses a simple oligomorphic decryptor. It is included for completeness, from the days before emulators were common. Its algorithm is:

```
p[i] = c[i] op k[j] (where op is - or ^, and j
indexes the buffer containing the decryptor itself)
```

A sample decryptor is:

```
0968:
   MOV         BX, CS
   MOV         DI, AX
   MOV         DS, BX
   NOP
   MOV         BX, 0008
   NOP
   MOV         SI, 0968
   CMP         BP, SI
   TEST        DL, BL
   MOV         CX, 0960
   CMP         BL, BH
   CLD
0980:
   MOV         DL, BYTE PTR DS:[SI]
   NOP
   SUB         BYTE PTR DS:[BX], DL
   INC         SI
   INC         BX
   CLD
   CMP         SI, 09A8
   JB          0993
   CMP         BL, CH
0993:
   MOV         SI, 0968
   CMP         BP, AX
   MOV         DI, AX
   LOOPW       0980
```

Zhengxi

Zhengxi (see [15]) uses a polymorphic decryptor in a long loop that includes dummy DOS and CP/M calls, making emulation a difficult prospect. Its algorithm is:

```
p[i] = c[i] op k (where op is a collection of + and ^)
```

A sample decryptor snippet is:

```
2550:
   CMP         SI, 41B0
   JBE         2561
   SBB        SI, 455F
   ADC        CL, BYTE PTR DS:[BX]
   CMP        CL, BYTE PTR CS:[BP + A48D]
2561:
   ADD        BX, 0096
   CLI
   ADC        CL, 1D
   AND        CX, SI
   CALL NEAR  24F7
   XOR        CX, 4994
   MOV        CL, B3
   XCHG       WORD PTR CS:[BX + 006D], DI
   CALL NEAR  24F7
   ADC        CL, 26
   ROL        CX, 01
   SUB        DI, DX
   JNL        258C
   XOR        CX, 3253
   CALL NEAR  24F7
258C:
   ADD        AX, 2161
   CALL NEAR  24F7
   JCXZ       2599
   XOR        CL, BYTE PTR CS:[BP + A622]
2599:
```

REFERENCES

[1] Tequila analysis, Richard Jacobs, ‘Cocktail of Viral Tricks’, *Virus Bulletin*, June 1991.

[2] Péter Ször, personal communications.

[3] IBM US patent 5,442,699 by Arnold, Chess, Kephart, Sorkin, White, ‘Searching for patterns in encrypted data’, from 15 August 1995.

[4] ‘The evolution of polymorphic viruses’, 1995.

[5] Mircea Ciubotariu, ‘Virus Cryptoanalysis’, *Virus Bulletin*, November 2003.

[6] Bruce Schneier, *Applied Cryptography*, 2nd edition, John Wiley & Sons, Inc.

[7] Igor Daniloff, RDA.Fighter analysis, ‘Fighting Talk’, *Virus Bulletin*, December 1997.

[8] Atli Gudmundsson, personal communications.

[9] Cormen, Leiserson, Rivest, *Introduction to Algorithms*, 2nd edition, McGraw-Hill.

[10] Peter Ferrie & Frédéric Perriot, Bagif analysis, ‘Looking a Bagift Horse in the Mouth’, *Virus Bulletin,* March 2003.

[11] Péter Ször, Drill analysis, ‘Drill Seeker’, *Virus Bulletin*, January 2001.

[12] Efish analysis, Peter Ferrie & Frédéric Perriot, *Virus Bulletin*, October 2004.

[13] Peter Ferrie, Magistr analysis, ‘Magisterium Abraxas, *Virus Bulletin,* May 2001.

[14] Adrian Marinescu, Perenast analysis, ‘Russian Doll’, *Virus Bulletin*, August 2003.

[15] Eugene Kaspersky, Zhengxi analysis, ‘Saucerful of Secrets’, *Virus Bulletin*, April 1996.