Cryptography and Viruses

Simone McCloskey
Math 187: Introduction to Cryptography
Spring 2005

Introduction

In the early days of personal computing, computer users lived in a fool's paradise. The first protocols for communication between computers included few provisions for security or data integrity; there was a certain amount of trust that came to exist between computer users simply because most of them were academics – scientists who saw these computers as tools for learning, not necessarily incapable of malicious exploitation in theory but certainly free of such applications in practice.

This paradise did not last long. As early as 1980 [7], people began to write malicious code for computers – ranging from viruses that replicated themselves due to user interaction, to the first network worm, the Morris worm, which took down a large chunk of the existing internet when it was released in 1988 (and got its author expelled from Cornell) [18].

With the new malicious code came new tools to detect it and stop it. Most of the initial viruses were amateurish, because they could afford to be; nobody expected the genesis of such programs and so there were no mechanisms in place to detect and disable them when they were written. Thus, the first attempts at antivirus programs were extremely simple; they knew what code was in existing viruses, they searched for that code within executable files, and if it was found, the computer had a virus. Most people still believe that such mechanisms are the extent of what is in place today to find viruses in antivirus software; however, crime will always evolve to beat the enforcers, and as soon as mass-market antivirus software such as Norton Antivirus, which debuted in 1990 [7], began using simple string detection to find and eradicate viruses, virus writers realized they had to step up their efforts to keep their code from being deleted before it had a chance to infect the systems they targeted. Only a year later, in 1991, the first polymorphic computer virus appeared in the wild – and so begins the history of the marriage of virus technology and cryptography.

This paper will focus on the history of the use of cryptographic methods in malicious code, most of which consists of extremely basic encryption, permutation and transposition, and information hiding that is reminiscent of the steganography used to hide data from humans – although it is different in that its goal is to evade computers. The technical detail will be minimal, but some familiarity with assembly programming may be helpful in reading the few code examples provided, as well as the discussions of architecture details. Other topics will include detection of polymorphic and metamorphic viruses and some discussion of how cryptographic methods are used to ensure integrity and security of data.

Polymorphism in Viruses

Polymorphism is the obvious application of cryptography that comes to mind when one contemplates how to use it in writing malicious software; if antivirus programs are on the watch for a certain string of code, it makes sense to use encryption to make the malicious code unreadable by anything but a computer that is already in the process of executing the virus by the time any decryption takes place. Indeed, as mentioned above, this use of encryption was found in the wild less than a year after the first mass-market antivirus programs. Furthermore, encryption of the virus body is relatively worthless when there is still a constant key and chunk of encryption code than other programs can search for; the point of polymorphism is to provide little to no constant virus content for scanners to latch onto [22]. This is implemented by encrypting the virus code with a different key each time (usually, this encryption is no more

complex than xor-ing the instructions with the key – although one author also advocates a Caesar cipher on the characters in the instructions [11], while another one claims that it is much more effective to layer many more simple encryptions on top of each other than just to use a single one, a sentiment reflected in the legitimate cryptography community as seen in the implementation of AES, for example [17]), and then attaching an extremely short encryption routine before the encrypted virus code in the infected file. This means that most strings that would have been constant in non-encrypted code are obscured in encrypted code, and leaves the antivirus software with only the short decryption code to search for.

More sophisticated polymorphic viruses get around the last limitation as well, by introduction of slight variations into the decrypt code, such as inserting a random number of do-nothing instructions or varying the registers used [3]. This is done by using system random number generators to determine numbers of iterations, initial indices, and so forth – and the decryption routines are generated as the encryption routine runs, guaranteeing matches in numbers while making the instructions appear random [20]. This obviously makes extensive use of operations that take immediate values as operands; however, even in an architecture that does not support immediate values, registers can also be randomized (anyway, the demand for viruses for non-x86 architectures is near-nonexistent). There will always be some short chunk of recognizable code, but the point of polymorphism is largely to make such chunks so small that the danger of false positives is too great for antivirus programs to alert users to their presence [22].

Once such viruses had been written and released, it was only a matter of time before the process of implementing polymorphism in viruses was automated. Several enterprising computer programmers realized that encrypting a virus was just like encrypting anything else, and could be done with a generic engine. Dark Angel, the author of the polymorphic engine DAME, includes sample code in his paper for multiple encryption and decryption routines that might be used in an engine, demonstrating that it is trivial to come up with different pieces of code that do the same thing when that thing is no more complicated than a simple loop of xors. He chooses to use a byte-by-byte encryption rather than using larger keys, which makes sense given that his polymorphism engine is for the x86 architecture, which has variable word length [8]. DAME was one of a number of polymorphic engines released around the same time which made it absolutely necessary that antivirus producers come up with a way to detect polymorphic viruses. They meant that people who knew nothing at all about encryption but could write decent non-encrypted viruses (or even use a prewritten virus code generator, which gives people who cannot even program in assembly language the basic tools to write their own viruses [3]) could write a normal virus and then utilize one of the many encryption engines available on the Internet to make it polymorphic, and therefore force detectors to step up their efforts with very little effort expended on the part of the virus author.

Once the engines had been released, the arms race between antivirus software companies and virus writers began. To detect polymorphic viruses, it was necessary to implement an operating system emulator so that software which no longer had established signature strings to grab onto could instead run the programs in a sandbox environment where they could do no harm to other files, but where they would hopefully decrypt themselves and reveal their original code, or at the very least do something suspicious, such as attempt to write in the boot sector or modify other programs. As a result, the next challenge to virus writers was to try to outsmart the emulator. At first, this was easy; early emulators only ran a few instructions of each program because of the speed limitations of personal computers at the time, so a virus author could add a

loop with a few thousand iterations at the beginning of the code and be assured that no antivirus program would ever read any actual instructions. Similar observations were made about other "features" of the emulators that could be exploited; most emulators did not emulate floating point operations, for example, because of speed limitations, or would exit after certain manually invoked interrupts, or would incorrectly interpret reads of default system registers [2], or overwrite stack values that the virus would normally be able to use (or, in some cases, be written to need to use, so that trying to run a virus in an emulator or other debugging environment would not only fail in detection, but also crash the computer as a bonus [14]). The key to avoiding detection by an emulator is to make sure the emulator never sees the virus code.

Modern computers generally have enough memory to allow antivirus programs to emulate floating-point instructions and long loops with little trouble, and support enormous pseudocode decision trees to identify suspicious behavior and compare it with that of known viruses. Interestingly, speculation about the future of traditional cryptography in malicious programs often brings up the prospect of viruses that may or may not be encrypted themselves, but that encrypt the data of the computer owner, threatening to destroy it or bar access to it if the virus is destroyed [1]. As high-speed worms gain popularity because of increased user awareness and antivirus use, as well as faster connections on more computers, the priority for many virus writers has moved away from evading detection and toward spreading fast enough that companies and individuals are infected before they have time to react. However, it is clear that encryption still has applications in this model – where criminal behavior and massive amounts of data are concerned, ingenious people will find a way to make obscuring that data to work for them.

Metamorphism in Viruses

A metamorphic virus rewrites its own code so as not to leave any recognizable string, which can be easily found by antivirus programs. It does this by permuting commands for which order does not matter, or by otherwise altering its own code from iteration to iteration. Early metamorphic viruses did little that was more involved than changing the registers they used on every reproduction, for example (which could be detected by a wildcard string, but not by the constant strings that were in general use in antivirus software that was used at the time, and so managed to escape detection). This is a method that uses simple substitution to obscure recognition; however, there are more sophisticated metamorphism schemes that use transposition as well, to far more devastating effect. For example, there have been viruses that reorder their subroutines between generations since the days of DOS, and these techniques can be replicated in Windows viruses. Any human reading the code would easily recognize what it did; however, most humans do not read the code that runs their computers, only virus checkers do. For fear of creating false positives, antivirus programmers are loath to pinpoint single lines of code as being part of viruses, since a single line of code says little about intent. This simple reordering was enough to foil early antivirus programs [9].

Virus writers write about both permuting code within sections, and also about peppering virus code with unconditional jumps so that it can actually be interleaved with the non-malicious code in the program that it infects [4]. Detection of the most complex viruses composed in this way is still an open research problem; if the virus is written so that no malicious command is more than a few instructions long, and it is impossible to predict where in the program the new lines will be, and if the virus overwrites existing code rather than adding to file size, it is almost

impossible to deterministically detect. Probabilistic methods are currently being explored, but they lead to more frequent false positives, which have been shown to irritate users of antivirus software [18]. In a paper discussing the concept of complete metamorphism in viruses, one virus writer suggests combining metamorphism with encryption in cases in which data cannot be altered, making constant search strings of any length almost impossible. However, the author also mentions that code with such complete metamorphism would grow prohibitively long relatively quickly (one of the necessary features of most viruses is that they are relatively small, so that they can be tucked into existing files or reside on the hard drive or in memory without raising a red flag), and proposes a code size constraint that would keep a lid on the size of the virus. Essentially, the article makes clear that there is a huge amount of overhead associated with writing malicious code that transforms completely to avoid detection; along with the malicious code itself, the virus writer needs to write code to control the transformations and more code to disguise that code, code to control the size of the code block, and code to encrypt things that simply cannot be altered. [15] The author concluded that if true metamorphism was to be implemented, he was unlikely to be the one to do it.

Another virus writer presents some rather half-formed ideas on using a context-free grammar to generate junk code to insert in viruses, which he admits appears to be a promising idea on its face but concludes is infeasible in real life because context-free grammars, once known, are recognizable by finite automata – and they themselves would have to be merely encrypted rather than morphed, making their eventual discovery inevitable [21]. However, the paper is notable in and of itself because it is an example of theoretical computer science subject matter directly influencing the thinking of malicious code writers. Many people dismiss virus writers as pranksters and vandals, but it pays to remember that many of these coders are far from amateurish; they work with the same corpus of knowledge as the people who write anti-virus programs to detect them, which is why the antivirus world so closely resembles an arms race much of the time. Virus writers represent a market for antivirus programs – in the virus writing world, it is silly not to test a new virus before releasing it, by pitting a few of the most popular antivirus utilities against it. To do so would be akin to releasing new software without an adequate testing phase; while it has certainly been done, it is not a prudent practice by any means.

Despite the theoretical limitations on size and sophistication of metamorphic viruses, implementations varying in sophistication have been seen in the wild. A Symantec whitepaper presents the following example of code generated by 2000’s Win32/Evol virus:

```
a. An early generation:

C7060F000055  mov      dword ptr [esi],5500000Fh
C746048BEC5151 mov     dword ptr [esi+0004],5151EC8Bh

b. And one of its later generations:

BF0F000055     mov     edi,5500000Fh
893E           mov     [esi],edi
5F             pop     edi
52             push    edx
B640           mov     dh,40
BA8BEC5151     mov     edx,5151EC8Bh
53             push    ebx
8BDA           mov     ebx,edx
895E04         mov     [esi+0004],ebx

c. And yet another generation with recalculated ("encrypted") "con-
stant" data.

BB0F000055     mov     ebx,5500000Fh
891E           mov     [esi],ebx
5B             pop     ebx
51             push    ecx
B9CB00C05F     mov     ecx,5FC000CBh
81C1C0EB91F1   add     ecx,F191EBC0h ; ecx=5151EC8Bh
894E04         mov     [esi+0004],ecx
```

In newer generations of this virus, even constant wildcard values change, and the Evol engine is also capable of inserting junk code in the virus body. This virus is impossible to detect with a search string, and unlike a virus that is merely encrypted, it does not have constant command sequences even in memory [9]. In short, the only way to recognize it is to run it in a virtual machine and recognize it by comparing it to known pseudocode – and other virus writers have been shown to come up with ways to foil algorithmic virus detection by exploiting weaknesses in the amount of memory such antivirus tools are able to use, and other oversights in writing the sandbox environments in which potential viruses are examined [2].

One of the most incredible examples of metamorphism is found in the virus Zmist, which Ferrie and Ször compare to the T-1000 from *Terminator 2* in its self-disguise capabilities. Even the virus's author was said to be surprised at how well it worked [9]. This virus actually decompiles executables and randomly intersperses its instructions throughout blocks of code (obviously, they are still organized so that they run, but the placement of blocks is not easily predictable). In this way, it obscures and randomizes the point at which its own code begins to execute. It alters its own instructions, reversing branch conditions, adding junk code, and substituting instructions for others that do similar things. The virus may or may not also polymorphically encrypt itself. Researchers for a major player in the antivirus industry admit that in coming years, viruses such as Zmist "will come very close to the concept of a theoretically undetectable virus". However, they are confident that ZMist itself is not a huge threat [10].

Metamorphic viruses are clearly a large part of the future of malicious code. The obvious weaknesses of polymorphic viruses make the less sophisticated among them easily detectable by antivirus software, whereas metamorphic viruses present a far greater challenge to those who would try to stop them.

Cryptographic Methods in the Hands of the Good Guys

Fortunately, computer criminals are not alone in having learned some basic cryptography. Security-minded computer users in all shades of hats use cryptographic ideas to ensure data integrity and secrecy. There are dozens of well-known applications that spring immediately to mind, such as SSL, SSH, e-mail encryption systems such as PGP and GPG, the digital watermarking used for digital rights management in most vendors of media files, wireless data encryption, and even uses as simple as password-protecting a course website. There are a few uses of cryptography for virus and malicious code spread prevention in particular that are worth a brief look.

Many systems currently provide support for cryptographic checksums, which are used to ensure that data has not been illicitly modified during transmission. For example, when downloading programs such as GNU Privacy Guard, users notice that the MD5 checksum of the file is available on the website. The purpose of this is that once the file is downloaded to the user's computer, the user can use a program to compute a cryptographic hash of the downloaded file, and compare that hash to the one listed on the website. Assuming that the hash is of high enough quality to ensure that the data has not been changed, and assuming also that the website hosting the code has not been compromised, the user can safely install the file with minimal worry about the file's intentions.

However, the assumption of a high-quality hash is rather dodgy. RSA is known to compute relatively good hashes, for example, but at the time that such systems were first recognized as necessary, it was slow enough that many computer users would balk at using it for checksum verification [6]. More commonly used is the MD5 hash, which has been discredited in recent years because it is easier to generate collisions in hashes than was previously supposed; regardless, it is still in use on the internet. Pseudocode to compute an MD5 hash is shown here:

```
//Note: All variables are unsigned 32 bits and wrap modulo 2^32 when
calculating

//Define r as the following
var int[64] r, k
r[ 0..15] := {7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22}
r[16..31] := {5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20}
r[32..47] := {4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23}
r[48..63] := {6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21}

//Use binary fractional part of the sines of integers as constants:
for i from 0 to 63
    k[i] := floor(abs(sin(i + 1)) × 2^32)

//Initialize variables:
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476

//Pre-processing:
append "1" bit to message
append "0" bits until message length in bits ≡ 448 (mod 512)
append bit length of message as 64-bit little-endian integer to message

//Process the message in successive 512-bit chunks:
for each 512-bit chunk of message
```

```
    break chunk into sixteen 32-bit little-endian words w(i), 0 ≤ i ≤ 15

    //Initialize hash value for this chunk:
    var int a := h0
    var int b := h1
    var int c := h2
    var int d := h3

    //Main loop:
    for i from 0 to 63
        if 0 ≤ i ≤ 15 then
            f := (b and c) or ((not b) and d)
            g := i
        else if 16 ≤ i ≤ 31
            f := (d and b) or ((not d) and c)
            g := (5×i + 1) mod 16
        else if 32 ≤ i ≤ 47
            f := b xor c xor d
            g := (3×i + 5) mod 16
        else if 48 ≤ i ≤ 63
            f := c xor (b or (not d))
            g := (7×i) mod 16

        temp := d
        d := c
        c := b
        b := ((a + f + k(i) + w(g)) leftrotate r(i)) + b
        a := temp

    //Add this chunk's hash to result so far:
    h0 := h0 + a
    h1 := h1 + b
    h2 := h2 + c
    h3 := h3 + d

var int digest := h0 append h1 append h2 append h3 //(expressed as little-endian)
```
[13]

It is clear that this is relatively simple to compute; certainly, it is far faster than RSA. This is given as an example program of exactly how a cryptographic hash might work; it is a fairly trivial set of operations. However, because of its collision vulnerabilities, there has been an attempt to move away from the MD5 hash in recent years. SHA-1 was believed to be more secure until February 2005, when researchers claimed that they had found collision vulnerabilities in it as well. The attack has not yet been published, but at the moment SHA-1 is still seen as more secure than MD5 ever was [19]. Regardless, as an assurance of data integrity, these hashing functions are useful but certainly not flawless. At the very least, checking the hash function can assure a downloader that the file has not been corrupted or altered in its transit between server and client; however, as a source of virus detection, these hashes are imperfect at best. If the source has been compromised, either through alteration of the website or through a collision attack, these hashes are of little use.

Another interesting possible use of hashing functions is in virus detectors. A 2003 paper suggests that rather than using memory-intensive and slow emulators to detect encrypted viruses, it might be more useful to compute a hash of suspicious code, using a quick function such as xor between lines, and look that hash up in a table that the antivirus program keeps current with

known hashes of virus code. If the hashes matched, the code would be identified as virus code and could be removed. If the hash did not match using xor, the antivirus program could also add, subtract, rotate and so forth, using all known ways that virus writers encrypt their code. This would make finding the key to a chunk of encrypted virus data completely unnecessary, since the hash could be found without it, and if the antivirus program managed to find the operation the virus used to encrypt, the hash would be the same regardless of the key used. If the antivirus program calculated hashes at runtime from already-known decrypted code, this method could even recover the key [5].

These are just a few of the intriguing ways that cryptographic hashing can be used to protect users from viruses. There are many other examples, such as systems that periodically calculate hashes of files to ensure that executables have not been altered without explicit cooperation from the computer user (an approach which has several unexpected drawbacks). Mac OS X includes a utility that allows users to encrypt their documents automatically so that if their computers fall into another person's possessions, their files will be useless without an administrator password. However, it is clear that virus writers have an easier time than the programmers attempting to stop them; there are myriad ways to infect computers, and the antivirus community too often finds itself in a position of reaction rather than prevention. Furthermore, encrypted files on a computer seized by law enforcement, for example, may be seen as more incriminating than human-readable files; recently, a man on trial in a Minnesota child pornography case had his possession of an e-mail encryption program used against him in court – although his hard drive did not even have any encrypted files stored! [12] A healthy amount of paranoia can work for or against a user depending largely on circumstances beyond her control. However, in using tried-and-true cryptographic techniques to try to safeguard and police data on personal and business computer, the antivirus community is taking steps in an extremely productive direction.

"Looking ahead to worm construction kits and metamorphic generators for worms is not as pleasant matter" is the delicate way that one researcher puts the general attitude of the security community [16]. Complete security from viruses is a distant fantasy in a real world in which there is money to be made from breaking into the machines of others. Virus writers have access to all the mathematical and computational tools that average users and antivirus writers have, and as is seen here, they make full use of at least the teachings of cryptography on a regular basis. This is only one set of examples of the sheer ingenuity and resourcefulness found in so many of the innovations of virus writers since 1980; it is a remarkably resilient community, and since the advent of spam and denial of service attacks, it has also become quite profitable to be a skilled virus author. These abuses of computer systems and cryptographic innovations are typical of a criminal underworld in which inventiveness, speed, and ability to recognize and grab opportunities are worth a huge premium. The only way antivirus writers can attempt to fight this kind of scourge is to know as much as they do about the methods they use and how to break them, and as the academic security community emerges to bridge the gap between industry and academic cryptographers, such goals are closer then they have ever been before. Nonetheless, virus detection and prevention will always be an uphill battle.

Sources

1. Balepin, Ivan. “Superworms and Cryptovirology: A Deadly Combination”. http://wwwcsif.cs.ucdavis.edu/~balepin/files/worms-cryptovirology.pdf.
2. Black Jack. “Anti-Heuristic Techniques”. http://madchat.org/vxdevl/papers/vxers/antiheur.html
3. Bontchev, Vesselin. “Future Trends in Virus Writing”. http://www.people.frisk-software.com/~bontchev/papers/trends.html. 1994.
4. Changeling. “Polymorphism Level 6B”. http://vx.netlux.org/lib/vch03.html. 1999.
5. Ciubotariu, Mircea. *Virus Bulletin*. “Virus Cryptoanalysis”. http://vx.netlux.org/lib/amc00.html. 2003.
6. Cohen, Fred. “A Cryptographic Checksum for Integrity Protection”. http://vx.netlux.org/lib/afc03.html.
7. “Computer Virus Timeline”. http://www.factmonster.com/ipka/A0872842.html. 2005.
8. Dark Angel. “Advanced Polymorphism Primer”. http://ftp.fortunaty.net/text/textfiles/virus/datut006.txt.
9. Ferrie, Peter and Peter Ször. “Hunting For Metamorphic”. http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf. Symantec White Papers, 2001.
10. Ferrie, Peter and Peter Ször. *Virus Bulletin*. “ZMist Opportunities”. http://pferrie.tripod.com/vb/zmist.pdf. 2001.
11. Twoflower, Jack. “Tricks to make your macro virus unscannable”. http://vx.netlux.org/lib/vjt01.html. 1999.
12. McCullagh, Declan. “Minnesota court takes dim view of encryption”. *CNET news.com*. http://news.com.com/Minnesota+court+takes+dim+view+of+encryption/2100-1030_3-5718978.html. 2005.
13. “MD5”. Wikipedia. http://en.wikipedia.org/wiki/MD5. 2005.
14. Midnyte. “An Introduction to Encryption, Part I”. http://vx.netlux.org/lib/vmn04.html. 1999.
15. Midnyte. “The Complete Re-write Engine”. http://vx.netlux.org/lib/vmn03.html. 1999.
16. Pearce, Stephen. “Viral Polymorphism”. http://vx.netlux.org/lib/asp00.html. 2003.
17. Rogue Warrior. “Guide to improving Polymorphic Engines”. http://vx.netlux.org/lib/vrw02.html.
18. Savage, Stefan. Lectures for CSE 127, Spring 2005.
19. “SHA Hash Functions”. Wikipedia. http://en.wikipedia.org/wiki/SHA-1. 2005.
20. Watson, Gary. “A Discussion of Polymorphism”. http://vx.netlux.org/lib/agw00.html. 1992.
21. Wintermute. “Polymorphism and Grammars”. http://vx.netlux.org/lib/vwm00.html. 1999.
22. Yetiser, Tarkan. “Polymorphic Viruses - Implementation, Detection, and Protection”. http://vx.netlux.org/lib/ayt01.html. 1993.