Trends In Computer Virus Research

by Dr. Frederick B. Cohen ‡

Abstract

In this paper, we discuss current trends in computer virus research. We begin with a quick review of the theoretical and practical history of viruses. Next we discuss recent results in the two major areas of current work; defending against malicious viruses; and designing useful viruses for efficient parallel processing. Finally we close by discussing future research topics.

‡ This research was funded by ASP, PO Box 81270, Pittsburgh, PA 15217, USA

1 Basic Theoretical Results on Computer Viruses

The "Computer Virus" problem was first described in 1984 [1], when the results of several experiments and substantial theoretical work showed that viruses could spread, essentially unhindered, even in the most 'secure' computer systems; that they could cause widespread and essentially unlimited damage with little effort on the part of the virus writer; that detection of viruses was undecidable; that many of the defenses that could be devised in relatively short order were ineffective against a serious attacker; and that the best defenses were limited transitivity of information flow, limited function so that Turing capability [2] was unavailable, and limited sharing.

In subsequent papers; it was shown that limited sharing, in the most general case, would cause the information flow in a system to form a partially ordered set of information domains [3]; it was proven that limiting of transitivity, functionality, and sharing were the only 'perfect' defenses [4]; and it was suggested that a complexity based defense against viruses might be practical [5]. It was also shown [4] that viruses could 'evolve' into any result that a Turing machine could compute, thus introducing a severe problem in detection and correction, tightening the connection between computer viruses and artificial life, and introducing the possibility that viruses could be a very powerful tool in parallel computing.

2 Malicious Viruses

While initial laboratory results showed that viruses could attain all access to all information in a typical timesharing computer system with properly operating access controls in only 30 minutes on average [1] and that network spread would be very rapid and successful [7], experiments and analytical results were severely limited by the unwillingness of the research community to even allow statistical data to be used in assessing the potential risks. [1] Although substantial theoretical results indicated how quickly viruses might be expected to spread given an accurate characterization of an environment [6], and an experiment at the University of Texas at El Passo showed that in a standard IBM PC network, a virus could spread to 60 computers in 30 seconds [7], actual spread rates could not be determined accurately until real-world attacks took place.

Real-world viruses started to appear in large numbers in 1987, when viruses apparently created in Pakistan, Israel, and Germany all independently spread throughout the world, causing thousands of computer systems to become unusable for short periods of time, hundreds of thousands of computers to display spurious messages, tens of thousands of users to experience denial of services, and several international networks to experience denial of services for short periods [7,8]. By 1988, there were about 20 well known and widely spread computer viruses, in early 1990, the IBM high integrity research laboratory reported over 125 unique viruses detected in the environment [9], and by March of 1991, between 200 and 600 unique real-world viruses were known in the research community [10], and over one new virus introduced into the global computing environment per day. [1]

In the period before viral attacks became widespread, there was little interest from the broader research community, and research results were considered of relatively little interest to funding agencies. Even though early results predicted many of the widespread implications we now see, very few organizations took any measures to defend themselves [11]. In the following years however, interest sprung up throughout the world research community, and there are now international computer virus conferences more than once a month, hundreds of university researchers, and tens of books on the subject. For more complete summaries of the field, the reader is referred to summary reports [19,20] and books [7,8] on the subject.

Most of the useful techniques for virus defense are based on basic results from fault-tolerant computing, with special consideration required to deal with defense against intentional attackers rather than random noise.

2.1 A Multitude of Broken Defenses

Many defensive ideas have been examined for their viability in virus defense. The vast majority of them have failed to pan out because there are generic attacks against them, they produce infinite numbers of false positives and false negatives, or they are too costly to be effectively applied [7]. We now examine several of the well known ideas that are in widespread use even though we have known for some time about their vulnerabilities.

The most common virus defense is the so-called 'scanner', which examines computer files to detect known viruses. Scanners have several important problems that have a serious impact on their current and future viability as a defense, most notably; [7,16] they only detect viruses known to the author, they produce infinite numbers of false negatives, they may produce false positives as new programs enter the environment, they are ineffective against many types of evolutionary viruses, they are not cost effective relative to other available techniques, and they become less cost effective as time passes.

There are a number of variations on scanners, most notably the so-called 'monitor' [28],

[1]Methods for making these counts are not standardized, and many 'new' viruses appear to be minor variations on previous viruses.

which is a variation on the 'integrity shell' technique described later in this paper [12,13], and which dramatically reduces the costs associated with detecting known viruses. [7,16]

Another interesting idea was the use of built-in self-test for detecting and possibly correcting viruses in interpreted information [5]. It turns out that all such mechanisms are vulnerable to a generic attack which is independent of the particular mechanism [7], but the concept of using complexity to make attack very difficult remains one of the most practical techniques.

A third common technique is to 'vaccinate' a program against viruses by modifying the program so that the virus is 'fooled into thinking' that the program is already infected. This has several very important drawbacks, primarily that not all viruses have to check for previous infection, vaccinating against large numbers of viruses may require so many changes that the resulting program will not operate, and n-tuples of competing viruses may make vaccination impossible [7].

Multiversion programming has also been suggested as a solution to the virus problem [1], and recent improvements in this technique have made this more and more feasible from an operational standpoint, [26] but the costs associated with these techniques make them tolerable only in a very limited number of environments [7], and it is unclear whether they will be useful in avoiding the effects of computer viruses because they don't address the ability to discern between legitimate and illegitimate changes. An n-version virus could presumably infect $(n+1)/2$ copies of the legitimate program, thus causing the voting technique to 'kick out' the legitimate program in favor of the virus [1].

2.2 Coding Techniques

Although precise virus detection is undecidable [4], we may be willing to suffer an infinite number of false positives and a very low probability of false negatives in order to have an effective defense. This can be achieved through the use of coding techniques which reliably detect changes. For example, a simple checksum or CRC code could be used to detect changes in files. The problem with many coding techniques is that they are easily forged, so that an attacker can easily make modifications which leave the code space unchanged [1]. The reason for this vulnerability is that many coding techniques are designed to detect corruptions due to random noise with particular characteristics, but they are not designed to detect malicious changes by intentional agents intent on bypassing them.

With a checksum which incorporates the size of a file, forgery is still usually straight forward because the file can be compressed before attack, leaving additional space that can be filled with null information or information selected to forge a valid checksum. A compression virus if this sort has been demonstrated [1].

In the case of a CRC coded checksum, attack is quite similar. Even if you don't know the order of the equation ahead of time, you can assume a large number of variables (as many as you have sample CRC codes and files for), and solve the equation. If there is enough data, all irrelevant coefficients will be determined as 0. If not, there is insufficient data for a unique solution. Several techniques have been devised to use multiple CRC codes with pseudo-randomly generated or user provided coefficients, but these appear to be simple to attack as well.

Another possibility is the use of information content measures. In this method, we calculate the information content of a data set using Shannon's method [30]. Unfortunately, for monograph content, it is trivial to switch two symbols without affecting the total content. Bigraph, trigraph, and higher order content can also be considered, but these do not appear to be significantly more difficult to forge. More generally, compression techniques can be used to decrease the overall content of a data set by flattening the probability distributions associated with symbols. [32] Once this has been done, symbols can be added to adjust the content and size of the data set. Computers also have finite precision, and a forgery needn't be exact, but only close enough for the precision being applied. Higher precision requires more time, and computing information content takes a substantial amount of time even at nominal precision.

The fundamental problem with all of these techniques is that they are designed to cover specific classes of changes, whereas an intentional attacker need not make changes in those classes in order to infect a file.

An alternative method designed to withstand substantial attack by knowledgeable attackers is the cryptographic checksum. The basic principle is to use a secret key and a good but fast cryptosystem to encrypt a file and then perform a checksum on the encrypted contents. If the cryptosystem is good enough, the key is kept secret, and the process meets performance requirements, the result is a usable hard-to-forge cryptographic checksum [5,14,15]. In this case, we can store the checksums on-line and unprotected, and still have a high degree of assurance that an attacker will be unable to make a change to the stored information and/or the associated cryptographic checksum such that they match under the unknown key when transformed under the hard-to-forge cryptographic checksum.

Fairly secure cryptographic checksums have been implemented with performance comparable to CRC codings. [14,15] In addition, the use of cryptographic checksums introduces the principle of the protection vs. performance tradeoff. In general, we can increase the difficulty of attack by increasing the cryptosystem key size, reducing the content per symbol (e.g. Huffman compression [32]), or improving our computational advantage through the use of a different cryptosystem [31]. Each of these normally involves more time for increased difficulty of attack, although changing cryptosystems may improve both protection and performance.

It is now apparent that we can use cryptographic checksums to reliably detect the changes associated with a computer virus even in the presence of a knowledgeable attacker, but we still have the problem of finding a way to efficiently apply the cryptographic checksum for virus detection.

2.3 Optimal Detection and Infection Limitation

We mentioned earlier that all built-in self-test techniques are vulnerable to a generic attack. The basis of this attack is that the virus could activate before the program being attacked, and forge an operating environment that, to the self defense technique, shows the altered information to be unaltered [7]. Since the introduction of this concept [27], several so-called 'stealth' viruses have appeared in the environment with the ability to forge unmodified files when the DOS operating system is used to read files, thus making detection by self-examination fail.

An alternative to built-in self-test is the use of a system-wide test capability that uses cryptographic checksums to detect changes in information. The question remains of how to apply this technique in an efficient and reliable manner. It turns out that an optimal technique for applying cryptographic checksums called an 'integrity shell' has been found [12,13].

This testing method performs tests just before interpretation. It is optimal [2] in that it detects all primary infection, [3] prevents all secondary infection, [4] performs no unnecessary checks, and we can do no better in an untrusted computing environment. [12]

Early experiments with integrity shells showed that they detected Trojan horses under Unix and gained rapid user acceptance in a programming environment [13]. More recently, cost analysis has also shown that this technique is more cost effective than other techniques in widespread use, including far less reliable methods such as virus scanners [7,16]. A similar cryptographic checksum method has been proposed for multi-level trusted systems [17], and finer grained hardware based detection at load time has also been proposed [18].

2.4 Automated Repair

[2]subject to the adequacy of the testing method C

[3]infection by information that contains a virus but has been trusted nonetheless

[4]infection by information infected through primary infection

Automated repair has been implemented with two techniques; for known viruses, it is sometimes feasible to remove the virus and repair the original data set with a custom repair routine; while general purpose repair is accomplished through on-line backups.

Although custom repair has some appeal, it is possible to write viruses that make this an NP-complete problem or worse through the use of evolution [1,4,7]. In several cases, customized repair has also produced undesired side effects, but this is primarily because of errors in identification of viruses or because certain side effects caused by viruses are not reversible from the information remaining in the data set. A simple example of an irreversible modification is the addition of instructions at a random location in the data space of a program. We can remove the instructions from the virus if we can find them, but we cannot necessarily determine what data they replaced. Similarly, a virus that adds bytes to a program to reach an aligned length and then adds the virus to the end, cannot be restored to the proper length because the proper length is no longer known.

As a general purpose repair scheme, on-line backups are used in an integrity shell to replace a data set with an image of the data stored when it was last trusted. This brute force method succeeds in all but the rarest cases, but has the undesirable side effect of doubling the space requirements for each covered data set. The space problem can be reduced by 50% or more in cases where original data sets have sufficient redundancy for compression to be effective, but the time and space overhead may still be unacceptable in some cases.

We can often implement on-line backups with no space overhead by compressing the original executable files and the on-line backups so that both require only 1/2 of the space that the original executable file required. This then slows processing at every program execution as well as at backup recovery time, and thus implements a slightly different time/space tradeoff.

On-line backups are also vulnerable to arbitrary modification unless they are protected by some other protection mechanism. Two such protection mechanisms have been devised; one cryptographically transforms the name and/or contents of the redundant data set so as to make systematic corruption difficult; and the other protects the on-line backups with special protection mechanisms so that they can only be modified and/or read when the integrity shell is active in performing updates and/or repairs. Both of these mechanisms have been quite effective, but both are vulnerable in machines which do not provide separate states for operating system and user resident programs (i.e. current personal computers).

A LAN based backup mechanism has also been implemented by placing backup files on the LAN file server. This mechanism has the pleasant side effect of automating many aspects of LAN based PC backup and recovery, which has become a substantial problem. In a typical LAN of only 100 computers, each with a failure rate of one failure per 2 years (i.e. a typical disk mean-time-to-failure for PC based systems), you would expect about 1 failure per week.

Some LANs have 10,000 or more computers, yielding an expected 100 failures per week. In these situations, automated LAN based recovery is extremely useful and saves a great deal of time and money.

Unfortunately, in many personal computers, the system bootstrap process cannot even be secured, and thus viruses can and have succeeded in bypassing several quite thorough integrity shell implementations. A recent development taken from fault tolerant computing [30] uses roll back techniques to 'SnapShot' system memory at bootup and perform a complete replacement of the system state with the known state from a previous bootstrap [25]. With this system, any memory resident corruptions are automatically removed at bootstrap and initial system testing can continue unhindered. The SnapShot mechanisms must of course be protected in order for this to be effective against serious attackers, but this dramatically reduces the protection problem and makes it far more manageable. In practice, this technique has been effective against all PC based bootstrap modifying viruses available for testing, and when combined with subsequent integrity checking and repair with on-line backups, results in a formidable barrier against attack.

2.5 Fault Avoidance Techniques

In almost all cases where viruses modify files, they exploit the operating system calls for file access rather than attempting to perform direct disk access. In systems with operating system protection, this is necessary in order to make viruses operate, while in unprotected systems, it is often too complex to implement the necessary portions of all versions of the operating system inside the virus, and it makes the virus less portable to hinge its operation on non-standard interface details that may not apply to all device types or configurations. An effective fault avoidance technique is to use enhanced operating system protection to prevent viruses from modifying some portion of the system's data sets.

It turns out that because viruses spread transitively, you have to limit the transitive closure of information flow in order to have an effective access control based defense [4]. In the vast majority of existing computer systems, the access control scheme is based on the subject/object model of protection [21], in which it has been shown undecidable to determine whether or not a given access will be granted over time. In an information system with transitive information flow, sharing, and Turing capability, this problem can only be resolved through the implementation of a partially ordered set [3,4].

To date, only one such system has been implemented [22], and preliminary operating experience shows that it is operationally more efficient and easier to manage than previous protection systems, primarily because it uses coarse grained controls which require far less time and space than the fine grained controls of previous systems, and because it has automated management tools to facilitate protection management. It has also proven effective against the transitive spread of viruses, thus confirming theoretical predictions.

Covert channels [23] still provide a method for attack by users in domains near the INF of the partially ordered set. Bell-LaPadula based systems [24] are vulnerable to the same sort of attack by the least trusted user [1,4,7], but with partially ordered sets, there needn't be a single INF, and thus even the impact of attacks exploiting covert channels can be effectively limited by this technique.

A 'BootLock' mechanism has also been devised to pre-bootstrap the computer with a low-level protection mechanism that masks the hardware I/O mechanism of the PC. BootLock provides low-level remapping of disk areas to prevent bootstrapping mechanisms other than the BootLock mechanism from gaining logical access to the DOS disk, and thus forces an attacker to make physical changes to a disk of unknown format or to unravel the disk remapping process in order to avoid phase p_2 protection. BootLock is also used to prevent disk access when the operating system is not bootstrapped through BootLock (i.e. from a floppy disk). Disk-wide encryption provides a lower performance but higher quality alternative to BootLock protection.

A wide variety of other fault avoidance techniques have been implemented, including testing of all disks entering an area for known viruses using a scanner [7], physical isolation from external systems [1], in-place modification controls for binary executables [29], and sound change control [7]. Although all of these techniques provide limited coverage against many current attacks, they have serious and fundamental cost and effectiveness problems that make them less desirable than more sound and cost effective techniques [7].

2.6 Defense-in-depth

As of this writing, the most effective protection against computer viruses is based on defense-in-depth. In this approach, we combine many approaches so that when one technique fails, redundant techniques provide added coverage. Combinations of virus monitors, integrity shells, access controls, virus traps, on-line backups, SnapShots, BootLocks, and ad-hoc techniques are applied to provide barriers against operation, infection, evasion, and damage by known and unknown viruses. [34]

In the laboratory and in operational experience, numerous experimental and real-world viruses have been tested against one such defense mechanism. Although most experiments indicate very little because their results are easily predicted, occasionally we find a surprising result and have to improve our models of what has to be covered and how to effectively cover it. The good news is that the technique of defense-in-depth tends to provide ample redundancy to withstand new attack mechanisms well enough to study the attack and improve the bypassed mechanisms. This is a vital point because with such a mechanism, we are now in a proactive posture, where defenders are not 'chasing' attackers, but rather attackers are 'chasing' defenders.

For example, the virus monitor is only effective against known viruses, and is thus quite weak. To avoid it, we only have to write a new virus or modify an existing virus in a non-trivial manner. This is done at a high rate, [5] so there is little realistic hope or desire for such constant updating. Since the time required for monitor operation increases linearly with the number of different viruses tested for, we decrease performance as we increase the known attack list. Based on experience, we select the most likely viruses and encode enough to cover over 90% of current attacks. [6]

The integrity shell detects all viruses which modify files unless they also modify the operating system mechanisms which the integrity shell uses to examine the files (A.K.A. a 'stealth virus') or circumvent the cryptographic mechanism. This covers over 99% of current known viruses, and in less vulnerable operating systems would probably be adequate on its own.

Access control has the effect of limiting the scope of the attack by preventing modification of non-writable files by the attack. To avoid this mechanism, it is necessary to either bypass its operation in memory or avoid the use of operating system calls entirely and perform purely physical disk access. This becomes quite complex as the number of different versions of the DOS operating system are quite high and hardware platforms vary considerably. In practice, only a few of the known viruses are able to bypass the access control mechanism (less than 1% of known viruses), and they do so by tracing operating system calls to locate internal addresses which they then directly access.

The series of virus traps which prevent damage by a variety of means are over 99% effective, but a skilled attacker can easily bypass these techniques.

The remapping of disk areas at bootup prevents over 99% of current automated physical attacks and the vast majority of manual attacks other than those performed by a skilled and well tooled operator with physical access to the hardware.

Finally, the SnapShot mechanism has never been circumvented, and as far as we can tell, can only be bypassed by intentional attack against the specific defensive mechanism. Thus it receives our 99% rating as well.

[5] 1 or more new viruses per day as discussed above

[6] 60 out of 600 known viruses currently represent over 90% of the attacks, so we get 90% coverage by testing for only 10% of known viruses.

A simple [7] calculation, assuming independence of mechanisms, is that the likelihood of successful attack is less than 1 in (90x99x99x99x99x99), or less than 1.2×10^{-12}! Unfortunately, protection just doesn't work that way, because we don't have random attackers. Most serious attackers will divide and conquer, bypassing each mechanism in turn. This requires a serious attacker to spend a substantial amount of time and effort. Thus, even though our probabilistic assumption is foolish, we have succeeded in 'raising the bar' high enough to fend off the vast majority of non-expert virus writers.

Another important issue to be understood in the case of viruses, is that selective survival assures us that as soon as one attacker succeeds in bypassing these mechanisms, the virus will spread and become available to far more attackers, who in turn may find other ways of exploiting similar weaknesses. Experience shows that attackers are intentional and malicious, and spend inordinate amounts of time finding ways to bypass protection mechanisms. Since no such defense is or can be perfect for the PC, we will perhaps always struggle with the problem of viruses, as long as we operate in this completely untrusted mode.

There are some other advantages and disadvantages of these mechanisms and we would be remiss if we did not point them out. In particular, general purpose mechanisms which are successful against intentional attackers tend to be quite successful against random events. On the other hand, as we provide defense-in-depth, performance suffers, and we may have to consider the extent to which we are willing to reduce performance in trade for coverage.

The integrity shell, automated recovery mechanism, access control mechanisms, SnapShot mechanism, and BootLock mechanism are all amenable to general purpose use in other protection applications, have a sound basis in theory for their effectiveness, and are attractive in other ways. Virus specific traps, monitors, trace prevention mechanisms, and other PC specific defenses are less portable and less applicable in other environments.

Performance can be greatly enhanced through hardware based implementations. To get an idea of the performance implications, the implementation we have been discussing typically operates in less than 3K of resident memory, and except for virus monitor and integrity shell operations requires only a few hundred extra instructions per affected operating system call. In practice, this has negligible performance impact. The monitor and integrity shell functions however take considerably more time because the operations they perform are considerably more complex. A typical virus monitor can check for 60 known viruses on a slow PC in under 1/2 second. If we expand to 600 viruses, this time exceeds one second, and as we start to examine harder to identify viruses, the time can go up by several orders of magnitude, depending on what we are looking for. A typical cryptographic checksum on a 20Mhz PC-AT (16 bit bus) with a 20msec hard-disk operates at 100Kbytes per second. The average DOS program in only about 17Kbytes long [13], so integrity shell operation slows

[7]but misguided

typical program startup by under 0.2 sec.

On-line backup restoration normally requires twice the time of program checking because for every read performed in checking, restoration performs a read and a write. With compression, this problem can be reduced, but only in trade for more computation time in the restoration process. We have never encountered a circumstance where it is preferable to not restore from on-line backups due to the time overhead, and considering that the time for restoration without on-line backups is at least several orders of magnitude longer, only space usage appears to be an impediment to the use of on-line backups.

The strength of this integrated set of redundant protection mechanisms is far stronger than a non-integrated subset because synergistic effects result in increased protection. As an example of synergy, with independent access controls and integrity checking, integrity checking information must be accessible to the attacker in order to be checked, and thus cannot be adequately protected by the access controls. Similarly, on-line backups cannot be protected from modification unless access control is integrated with automated repair. Memory limits of DOS cause the size of resident memory usage to be a critical factor in protection as well. By combining mechanisms we dramatically reduce resident memory requirements, which is another synergistic effect. There are many other synergistic effects too numerous to list here.

Ultimately, a sufficiently motivated, skilled, and tooled attacker with physical access to a system will bypass any protection mechanism, but in the case of computer viruses, highly complex mechanisms are more likely to require large amounts of space and time and be noticed because of their overall system impact. If we can drive the complexity of automated attack high enough without seriously impacting typical system performance, we will have achieved our primary goal.

2.7 Architectural Implications

As we have seen, software based virus protection in untrusted computing environments depends heavily on software based fault-tolerant computing. Not only do we require defense-in-depth in order to be effective, but we often have to use redundancy within each method to assure reliability of mechanisms against attackers. Although these software techniques are quite effective at this time, ultimately a hardware supported solution is far preferable.

In the field of computer architecture, information protection has historically been fundamental. Many major advances in information protection have led directly to new architectural structures in computers, and by this point in time, about 25% of the hardware in most modern CPUs is in place for the purpose of protection. At the hardware level, fault tolerant computing has gained legitimacy as a field which studies integrity of hardware structures, but at the systems level, it has failed to protect against software based attacks. Thus information protection at the hardware level has a firm grip in most computer engineering programs, but exists under a name that hides its relationship to other information protection subfields like 'computer security' and 'cryptography'. Computer viruses have demonstrated the extreme lack of integrity in modern systems and networks, and have demonstrated the short sightedness of our research community in ignoring integrity, a major protection issue.

A large portion of the applications for 'super-computers' are in the code breaking area. Encryption hardware is a substantial portion of the military budget, and nearly every modern mainframe computer system has either hardware or software encryption capabilities. Every automatic teller machine uses encryption hardware and software, as do most point of sale terminals, and many credit card checking machines. A major complaint against the personal computer has been its lack of protection hardware, which causes crashes and numerous other problems. The major differences between the first generation of personal computers and those coming out now are the addition of protection hardware and improved performance.

Introducing standard hardware-based operating system protection provides dramatic improvements in access control and separation of operating system functions from application program attacks, but this alone does not resolve the virus problem. For example, the first virus experiments were performed on a Unix based system with hardware based operating system protection. They did not exploit any operating system properties other than the sharing and general purpose function, [1] and they demonstrated the ability to attain all rights in under 30 minutes on the average.

For integrity shells and virus monitors, we can save considerable amounts of time by incorporating the checking mechanism in the operating system, since checking a program and then loading it duplicates I/O operations. Hardware based implementation yields several orders of magnitude in performance which can be used to improve performance and/or difficulty of attack. The most advantageous hardware location for generating and testing these codes is in the disk controller, where other checks such as parity and CRC codes are done. The disk controller could easily return these codes to the operating system as a result of DMA transfers, and the operating system could then combine the sequence of codes generated for a file to yield a cryptographic checksum.

Another alternative for highly trusted operating systems is maintaining only the modification date and time of files and a list of the last authorized modification date and time [12]. The operation is essentially the same as an integrity shell, except that we must have an operating system that maintains file dates and times as reliably as cryptography covers changes. In systems without physical access and very good operating system controls, this is feasible, but no current system meets this standard. For example, current systems allow the clock to be changed, which completely invalidates this mechanism. Updating file modification dates would also invalidate this mechanism. Another advantage of the cryptographic checksum is that it operates properly between machines and across networks. Since the cryptographic checksum is simply a mathematical function of the key and file, no hardware dependencies need be involved.

We must also be careful to assure that the mechanism of updating checksums does not become too automated. If it simply becomes a check for disk errors, it will not fulfill its purpose of controlling the propriety of change. After all, the legitimacy of change is a function of intent [12], and if we automate to the point where people do not specify their intent, we return to the situation where a virus can make a seemingly legitimate change.

Other mechanisms that should be retained even when operating system protection is facilitated by hardware is the BootLock and SnapShot mechanisms. These mechanisms are vital in assuring that the bootstrapping process has not been corrupted. In every case we are aware of, this is feasible given an attacker with physical access to the system, adequate tools for system debugging, and adequate knowledge and persistence.

3 Benevolent Viruses

On March 22, 1991, the world high speed computing record was broken by a Massachusetts company specializing in parallel processing. The previous record holder, contrary to popular belief, was a computer virus written and distributed in the Internet, one of the World's largest computer networks, by Robert Morris, then a graduate student at Cornell University.

The 'Internet Virus' was a relatively small computer program written in the 'C' programming language. It was designed to replicate itself in computers networked to Mr. Morris's and use the processing and communication capabilities of these computers to spread to other networked computers. [35] Because of a fundamental design flaw, the Internet Virus spread too quickly, and its exponential growth caused widespread denial of services to Internet users over a two day period. Eventually, Robert Morris, like his virus, was caught. Mr. Morris was tried and convicted of unauthorized access to 'Federal Interest Computers', and to pay for his transgression, he had to pay a fine and perform community service, and was kicked out of graduate school, [36] but his virus stands as a startling example of the potential of computer viruses for both malicious and beneficial purposes.

The design flaw that caused unchecked growth demonstrates one of the malicious aspects of this virus, and unfortunately, many of the other computer viruses we hear about are also malicious, but like any new technology, viruses are a two edged sword. Consider that the

Internet Virus performed about 32 million operations per second on each of 6,000 computers, and another 3.2 million operations per second on each of 60,000 computers, for a grand total of 384 Billion operations per second! It took hundreds of person-years of work and millions of dollars to design the computer hardware and software that broke this processing record, while the Internet Virus was written by one graduate student using existing computers in his spare time over a period of a few months. For pure processing cycles, computer viruses are some of the fastest distributed programs we know of, but unfortunately, we haven't yet grown enough scientifically or ethically to exploit their vast potential.

The same issues that make viruses a serious threat to computer integrity [7] make them a powerful mechanism for reliable and efficient distribution of computing. They distribute freely, easily, and evenly throughout a computing environment; they provide for general purpose computerized problem solving; and they are very reliable even in environments where computer systems fail quite often.

Efficient uniform distribution of computing between computers working together on the same problem is one of the hardest problems we face in parallel processing. For large parallel processors working on complex problems, it is often more difficult to find an optimal way to distribute problem solving among available computers than it is to do the problem solving once the processing is distributed. With computer viruses, we automatically get distribution based on available processing for many computations because computers with less available processing tend to be slower to replicate viruses, while computers with more available processing tend to provide faster replication. Since viruses can spread wherever information spreads and is interpreted, viruses can eventually distribute themselves throughout networks regardless of how the computers are interconnected as long as they have general purpose function, transitive communication, and sharing [1,4]. These two features eliminate the need to spend time figuring out how to distribute processing across computers in many applications.

Although general purpose problem solving is rarely a problem in computers today, reliability is particularly critical to large parallel processing applications because as the number of computers involved in problem solving increase, the likelihood of a failure during processing also increases. [33] The Internet Virus continued processing even though many systems in the Internet were turned off and many subnetworks were disconnected in an effort to stop it. Few modern parallel processing applications could continue processing in this sort of environment. In fact, most parallel processing computers today could produce erroneous results without even producing an error message, much less processing correctly when some of the computers fail. Viruses on the other hand have inherent reliability because of their ability to replicate and spread. Most of the computer viruses we know about work correctly in a wide variety of computer makes and models, work in both networks and isolated systems, work in many different versions of operating systems, spread to backup tapes and are revived when backups are restored, work on floppy-disks, hard-disks, and network pseudo-disks, and survive system failures and reconfigurations. Some of them even operate across different operating systems and types of computers. [7]

Efficient and reliable distribution of processing in itself is not enough for efficient problem solving. For some problems, we have to be able to communicate results between processing components, while in other problems the time required for processing may be too small to justify the time required for distribution. The problem of controlling virus growth must be addressed before widespread use of viruses in existing computer networks will become acceptable to the user community, and evolution of viruses over time will probably be a vital component to their long term utility. Many issues in viral computation are not yet resolved, but there is also a substantial body of knowledge and experience to draw from.

The use of self-replicating programs for parallel processing is not new. In fact, John vonNeumann, one of the pioneers of the computer age, described reliable self-replicating programs in the 1940s. [37] In many early works, the 'living' computer program was not just a distant possibility, but the intent of the exercise. Early papers talked of 'making reliable organisms out of unreliable organs', and 'intelligent self-organizing systems with numerous components'. [38] Over the last 50 years, many authors have reported isolated experiments, and slow progress has been made.

3.1 The Worm Programs, Computer Viruses, and Artificial Life

In 1982, a series of successful experiments with parallel processing using self-replicating programs that spread through the Xerox computer network. [39] To quote this paper:

> "A *worm* is simply a computation which lives on one or more machines. ... The programs on individual computers are described as the *segments* of a worm ... the worm mechanism is used to gather and maintain the segments of the worm, while actual user programs are then built on top of this mechanism."

These so-called 'worm' programs would install segments on computers which were not in use, performing 'segments' of the parallel processing problem being solved. Whenever a user wanted to use their computer, they simply pressed the reboot button, and normal operations resumed. During the day, the worm would be trimmed back, running only on a few computers that were not in use, but at night, it would become active all over the network, performing tens of millions of useful calculations per second. Unfortunately, an error in one copy of the worm ended their experiments by causing the global Xerox network to reboot to the worm instead of the normal operating system. The entire network had to be restarted. A number of worm programs were also run on the Arpanet during the 1970s, some of them even capable of limited replication. For unspecified reasons, worm researchers apparently stopped performing these experiments in the early 1980s.

In 1984, the first experiments with 'Computer Viruses' as we know them today were performed. [1] To quote this paper:

> "We define a computer 'virus' as a program that can 'infect' other programs by modifying them to include a possibly evolved copy of itself."

These 'Viruses' had many implications for integrity maintenance in computer systems, and were shown to be quite dangerous, but their potential for good was also introduced. A practical virus which reduced disk usage in exchange for increased startup time was described, and this technique that is now commonplace in personal computer systems. A formal definition for viruses, which for mathematical reasons encompasses all self-replicating programs and programs that evolve and move through a system or network, was first published in 1985. [4] This encompassed many of the worm programs under the formal umbrella of computer viruses. This work also pointed out the close link between computer viruses and other living systems, and even melded them into a unified mathematical theory of 'life' and its relationship to its environment. These experiments were terminated rather forcefully because they were so successful at demonstrating the inadequacy of contemporary computer security techniques, that administrators came to fear the implications.

In the mid 1980s, Scientific American began publishing a series on a mathematical 'game' called 'core wars' [40], in which two or more competing programs struggled for survival in a simulated computer, while the game of 'Life' which simulates 'living cells' in a cellular automata has existed for quite a long time [38]. To the extent that they replicate and/or evolve within the environment, they meet the mathematical definition of a computer virus. These examples of viruses have met with no significant resistance in the research community, presumably because they have no widespread impact. The same cannot be said for other experiments.

In 1986, the first experiment with a PC based network virus was performed by several graduate students at the University of Texas at El Passo, who found that the virus spread to 60 computers in 30 seconds. These experiments were quickly terminated because they were so successful that researchers feared the consequences if the viruses escaped [7]. In 1987, a graduate student researcher at Penn State University was forced to terminate experiments because of fears from members of the university community, even though there were no problems related to this research. In early 1988, a professor at the University of Cincinnati was forced off the University computer systems because a systems administrator saw the word 'virus' in the professor's computer account and decided it was too much of a risk to allow that sort of work. These ‘knee jerk’ responses have made the road for legitimate research on the beneficial implications of viruses very difficult, but despite these encumbrances, the research has continued.

In 1987, the first ‘Artificial Life’ conference was held, with researchers gathering from around the world to present papers on a loosely knit combination of many independent research areas that seek to describe, simulate, analyze, or implement living or life-like systems. [38] In recent years artificial life has received increased attention both in the popular and the scientific communities, partly because of the emergence of computer viruses in modern computer networks, partly because of the growing number of interested researchers with results to report, and partly because of the efforts of some members of the research community who have started to make communication between divergent fields easier by providing common venues for publication and interaction.

giving some basic background, this discussion leads us to ask some burning questions about the practicality of computer viruses for computation and the tradeoff between the potential benefits and the potential harm of using viruses for this purpose. In the remainder of this discussion, we will review some recent examples of practical computer viruses, some of the deep issues that come up in this research, and grapple just a bit with the issue of balancing the good with the bad.

3.2 The Viral Bill Collector

One useful computer virus is the automated ‘bill collector’, a virus that operates in a specially designed computing environment. Two such bill collectors have been implemented under the Unix operating system, one for a small business with thousands of orders of only a few hundred dollars each, and the other for a law office that specializes in collecting unpaid commercial loans for thousands of clients with tens of thousands of debtors. Taking the law office as an example, the computing environment ‘births’ a new bill collector every time a new case is entered by a user, kills bill collectors whenever the user indicates that a bill is fully paid or the case is abandoned, and allows the bill collector to write letters and evolve through its life-cycle. In its simplest form, each bill collector is just a small program that collects a single bill by sending a series of letters over time depending on real-world events or the lack thereof as indicated by the user.

It turns out that writing a computer program to collect a single bill is not very hard to do. In fact, in a few hours, an average programmer can write a simple bill collecting program that will do a pretty good job of collecting a single bill, based on the methods of collection used by an expert human bill collector.

If we were writing a high volume collection system in the standard fashion, we would implement a complex database management system. Next, we would devise a technique for scanning through this database periodically to determine which collection cases required action at any given time, and based on this list, have the bill collection program perform collections on all applicable cases. To collect statistics and resolve the status of cases, we would then have to implement another database scanning system, and the list goes on and on. By the time we are done, we have a very large and complex system.

With the viral programming approach, we take another tactic altogether. Instead of creating a large centralized bureaucracy which controls and directs all activities, we distribute all functions to the individual bill collectors. Each bill collector only has information related to its own collection case and the ability to selectively call upon its various scenarios for bill collection. Instead of scanning a database for bills to be collected, each bill collector schedules a 'wake up' call for the next time it has to do something. If some outside activity like a payment or a response to a previous action takes place, the human operators 'wake up' the appropriate bill collector by sending it the new information. The collector then reacts to the situation by 'evolving' its state and line of pursuit to meet the new situation, reschedules its pending wake up calls, and goes back to sleep.

One major advantage of writing a simple virus for this task is that all we have to do is provide basic replication, evolution, and wake up call mechanisms, and we don't have to deal with the complexity of large databases or long database searches to determine when to do what. Another major advantage is that since we are running many very small and independent programs instead of one large program, we can more easily distribute the computing load over a multitude of machines, and we don't have to deal with issues like simultaneous access, file locking, and process blocking. There are also disadvantages, in that collecting data from all of the bill collectors is somewhat less efficient than looking up all of the data in a common database, and making changes to all of the independent bill collectors requires a systemic evolution process.

To collect global data with a viral bill collector, we typically awaken every bill collector, ask for the required information, and collate the results. We end up writing several such data collection applications which we then schedule for operation in off hours. The results from the previous day are then always available the next morning, and since we are usually using otherwise idle computer time during off-peak hours, the inefficiency is not unduly bothersome. In an emergency, we can awaken all of the bill collectors during the day to get more instantaneous results, but in this case system wide performance suffers greatly. This feverish sort of activity is only required in the rarest of circumstances.

The advantages of the viral approach are even clearer in a networked environment. In a network, viruses self-distribute efficiently by replicating in remote processors. We can use performance as a survival criterion, or simply distribute performance information to viruses about to replicate in order to help balance the load. There are no shared data problems as in a distributed database because all of the data associated with each virus is local to that virus. When global data is requested, we get maximum parallelism for a significant portion of the process because the only bottleneck is in the reporting of results, which in the case of the bill collector takes relatively little bandwidth.

To evolve all of the independent bill collectors in response to new information requirements, we again awaken every bill collector, provide the necessary information for it to change its programatic (as opposed to genetic) codes, and allow it to return to sleep. Just as in the global data collection case, we can perform this sort of change during off hours, and there is rarely a case where we need such a change on a moment's notice. We needn't use this sort of systemic evolution all of the time. Instead, we can take the less invasive approach of developing better and better bill collectors over time, and simply birthing them for collecting new bills. Eventually older bill collectors die out as they conclude their tasks.

Our first viral bill collector was implemented by one programmer in one week, in 1986. It has evolved successfully in response to new needs without any global changes since that time, and although this sort of evolution requires human design, the amount of work is minimal. The bill collection viruses also coexist in the environment with a set of 'maintenance' viruses that periodically awaken to perform cleanup tasks associated with systems maintenance.

3.3 Maintenance Viruses and the Birth/Death Process

Maintenance viruses, as a class, seem to be one of the most useful forms of computer viruses in existence today. Put in the simplest terms, computer systems are imperfect, and these imperfections often leave residual side effects, such as undeleted temporary files, programs that never stop processing, and incorrectly set protection bits. As more and more of these things happen over time, systems become less and less usable, until finally, a human being repairs the problems in order to continue efficient processing.

In the case of the viral bill collector, the design of the system is such that temporary files are stored under identifiable names, processing times for each bill collector tends to be relatively short, and protection bits are consistently set to known values. To reduce manual systems administration, we decided to implement viruses that replicate themselves in limited numbers, seek out known imperfections, and repair them. Over time, we reduced systems administration to the point where the viral bill collector operated for over two years without any systems administration other than adding and removing users. The maintenance viruses were so successful that they even removed side effects of other failed maintenance viruses.

To assure the continued survival of the maintenance viruses, they are born with a particular probability every time a user awakens a bill collector, and to assure they don't dominate processing by unbounded growth, they have limited life spans and replicate with lower probability in each successive generation. With proper probabilities, this combination of factors successfully produces stable populations of maintenance viruses and is quite resilient.

These "birth/death" processes are central to the problem of designing viruses that don't run amok, as well as to the evolution of viral systems over time. If it weren't for the death of old bill collectors and maintenance viruses, the system would eternally be collecting bills and performing maintenance under old designs, and the number of bill collectors and maintenance viruses would grow without bound. A global modification of all of the existing bill collectors would be required to make a system change, and this might be very hard to accomplish in a complex network. Birth and death processes appear to be vital to optimization in viral systems where the environment changes dramatically with time, since what is optimal today may not even survive tomorrow.

3.4 Toward Random Variation and Selective Survival

We have spoken of evolution, but to many, this concept doesn't seem to apply to computer programs in the same way it applies to biological systems. In its simplest form, we speak of systems evolving through human reprogramming, and indeed, the term evolution seems to accurately describe the process of change a system goes through in its life cycle, but this is only one way that programs can evolve.

Consider a bill collector that uses pseudo-random variables to slowly change the weighting of different collection strategies from generation to generation, and replicates individual bill collectors with a probability associated with its profitability (the net fee collected after all expenses of collection). In this case, assuming that the parameters being varied relate to the success of the collection process in an appropriate manner, the 'species' of available viruses for the collection process will seem to 'evolve' toward a more profitable set of bill collectors.

If we use less variation on bill collectors that are more successful, we may tend toward local optima. To attain global optima, we may occasionally require enough randomness to shake loose from local optima. Over time, we may find many local optima, each with a fairly stable local population of bill collectors. Thus different species of bill collectors may coexist in the environment if there are adequate local niches for their survival. Cross breeding of species is feasible by taking selected parameters from different species to birth new bill collectors. Some will thrive, while some will not even survive. As the external environment changes, different species may perform better, and the balance of life will ebb and shift in response to the survival rates. This evolutionary process is commonly called "random variation and selective survival", and is roughly the equivalent of biological evolution as we now commonly speak of it.

3.5 Complex Behaviors, Generating Sets, and Communication

The behavior we are discussing is getting complex, involving local and global optimization, evolution over time, and even the coexistence of species in an environment; and yet the computer programs we are discussing are still quite simple. Our viral bill collectors consist of only a few pages of program code, and yet they perform the same tasks carried out by much larger programs. The inclusion of evolution in experimental systems has been accomplished in only a few lines of program code. [1] We create a small "generating set" of instructions which creates a complex system over time through birth/death processes, random variation and selective survival, and the interaction of coexisting species in the environment. Even quite simple generating sets can result in very complex systems. In fact, in many cases, we cannot even predict the general form of the resulting system. [4]

Our inability to accurately predict systemic behavior in complex systems stems, in general, from the fact that it is impossible to derive a solution to the halting problem. More specifically, it was proven [4] that a virus can evolve in as general a fashion as a computer can compute, and therefore that the result of viral evolution is potentially as complex as Turing's computation. It seems there is little we can do about predicting the behavior of general purpose evolutionary systems, but just as there are large classes of computer programs with predictable behavior, there are large classes of evolutionary systems with predictable behavior. Indeed, in the same way as we can generate computer programs from specifications, we can generate evolutionary systems from specifications, and assure to a reasonable degree that they will act within predefined boundaries. In the case of the maintenance virus, we can even get enhanced system reliability with viral techniques. Unfortunately, we haven't yet developed our mathematical understanding of viruses in an environment to the point where we can make good predictive models of these sorts of systems, but there is a general belief that many important problems are not intractable at the systemic level, and if that is true, we may be able to make good predictive models of the behavior of large classes of useful viral systems.

One of the ways we can design predictable viral systems is by adding communications. Completely deaf and dumb viruses have a hard time surviving because they tend to be born and die without any controlling influences, and we get unstable situations which either consume too many resources and ruin the ecology or die from lack of sufficient biomass. With even rudimentary communications, viruses seem to survive far better. For example, most real-world computer viruses survive far better if they only infect programs that are not yet infected. Too much communication also makes viruses inefficient, because they have to address an increasingly global amount of information. We suspect that communications is beneficial to viral survival only to the extent that it helps to form stable population relative to the resources in the environment, but in terms of designing predictable viral systems, communications seems to be key.

The maintenance viruses described earlier provide a good example of viral communication. There is no 'direct' communication between the maintenance viruses, but they end up communicating in the sense that what each virus does to the environment alters the actions of other viruses. For example, a maintenance virus that deletes temporary files that haven't been accessed in 24 hours or more will not delete any files that a previous maintenance virus has already deleted, since the earlier virus already consumed them. Since the latter virus acts differently based on the actions of the earlier virus, there is a rudimentary form of communication between the viruses via changes in the environment. In fact, humans communicate in much the same way; by making changes in the environment (e.g. sound waves) that affect other humans. The net effect is that maintenance viruses act more efficiently because they rarely interfere with each other.

3.6 Toward Widespread Viral Computation

The possibilities for practical viruses are unbounded, but they are only starting to be explored. Unfortunately, viruses have gotten a bad name, partly because there are so many malicious and unauthorized viruses operating in the world. If the computing community doesn't act to counter these intrusions soon, society may restrict research in this area and delay or destroy any chance we have at exploiting the benefits of this new technology. There are now many useful tools for defending against malicious viruses and other integrity corruptions in computer systems, and they can often be implemented without undue restriction to normal user activity, but perhaps another tactic would also serve society well.

The tactic is simple; instead of writing malicious viruses, damaging other people's computer systems, hiding their identity, and risking arrest, prosecution, and punishment; virus writers could be provided with a legitimate venue for expressing their intellectual interest, and get both positive recognition and financial rewards for their efforts. By changing the system of rewards and punishment, we may dramatically improve the global virus situation and simultaneously harness the creative efforts of virus writers for useful applications. One instance of such a tactic is the 'Computer Virus Contest' [41], which gives an annual cash prize for the most useful computer virus submitted. The contest rules prohibit the use of viruses that have been released into uncontrolled environments, viruses placed in systems without explicit permission of the owner, and viruses without practical mechanisms to control their spread.

4 Summary, Conclusions, and Further Work

We have described a substantial set of redundant integrity protection mechanisms used in defending against computer viruses in untrusted computing environments. They include applications of coding theory, cryptography, operating system modifications, redundancy for detection and repair, fault avoidance, synergistic effects, and defense-in-depth.

These protection measures comprise one of the most comprehensive applications of software-based fault-tolerance currently available, and their widespread use represents a major breakthrough in the application of software-based fault-tolerance. They have proven effective against a wide range of corruption mechanisms, including intentional attacks by malicious and intelligent agents. Some of these techniques are used for virus defense in over 100,000 systems, there are something like 10,000 systems currently exploiting all of these techniques in combination, and in the next few years, the total number of systems protected by these mechanisms are expected to exceed 1,000,000.

The protection measures discussed herein are effective to a large degree against computer viruses, but the implications of this work on high integrity computing are far broader than just defense against viruses or even 'computer security'. The integrity shell, for example, detects large classes of corruptions, including single and multiple bit errors in storage, many sorts of human and programmed errors, accidental deletion and modification, transmission errors in networking environments, and read/write errors in unreliable media such as floppy disks. SnapShot techniques have widespread applications for high integrity bootstrapping, and similar techniques are already used in limited ways for error recovery in other areas.

Improvements based on hardware implementation will provide dramatic performance and reliability enhancement, as will the application of these techniques in systems which already exploit hardware based operating system protection.

A number of related research areas are already being pursued. The highest priorities at this time are being given to enhanced performance, improvements in evolutionary codes to make automated bypassing of protection mechanisms very complex, other exploitations of computational advantage in driving the complexity of attack up without seriously impacting the performance of the defenses, improved techniques for exploiting hardware based protection, and the application of these techniques to other operating systems and architectures.

The emerging computer virus technology, like all new technology, is a two edged sword. Just as biological viruses can cause disease in humans, computer viruses can cause disease in computer systems, but in the same sense, the benefits of biological research on the quality of life is indisputable, and the benefits of computer virus research may same day pay off in the quality of our information systems, and by extension, our well being. To the extent that we can learn about our ecosystem by studying informational ecosystems, we may save ourselves from the sorts of mistakes we have already made in dealing with our environment.

Somebody once said that computer systems are one of the greatest laboratory facilities we have. Their ability to model and mimic life and life-like situations is astounding both in its accuracy and in its ability to allow experiments that would be unconscionable or infeasible in any other laboratory. We have the unique opportunity to use this laboratory to get at the fundamental nature of living systems, if we can only get past our biases and get on with the important work awaiting us.

5 References

- 1 - F. Cohen, “Computer Viruses - Theory and Experiments”, originally appearing in IFIP-sec 84, also appearing in DOD/NBS 7th Conference on Computer Security, and IFIP-TC11 “Computers and Security”, V6(1987), pp22-35 and other publications in several languages.
- 2 - A. Turing, “On Computable Numbers, with an Application to the Entscheidungsproblem”, London Math Soc Ser 2, 1936.
- 3 - F. Cohen, “Protection and Administration of Information Networks with Partial Orderings”, IFIP-TC11, “Computers and Security”, V6#2 (April 1987) pp 118-128.
- 4 - F. Cohen, “Computer Viruses”, Dissertation at the University of Southern California, 1986.
- 5 - F. Cohen, “A Complexity Based Integrity Maintenance Mechanism”, Conference on Information Sciences and Systems, Princeton University, March 1986.
- 6 - W. Gleissner, “A Mathematical Theory for the Spread of Computer Viruses”, “Computers and Security”, IFIP TC-11, V8#1, Jan. 1989 pp35-41.
- 7 - F. Cohen, “A Short Course on Computer Viruses”, ASP Press, PO Box 81270, Pittsburgh, PA 15217, 1990.
- 8 - H. Highland, “Computer Virus Handbook”, Elsevier, 1990.
- 9 - S. White, “A Status Report on IBM Computer Virus Research”, Italian Computer Virus Conference, 1990.
- 10 - K. Brunnstein, “The Computer Virus Catalog”, DPMA, IEEE, ACM 4th Computer Virus and Security Conference, 1991 D. Lefkon ed.
- 11 - F. Cohen, “Current Trends in Computer Virus Research”, 2nd Annual Invited Symposium on Computer Viruses - Keynote Address, Oct. 10, 1988. New York, NY
- 12 - F. Cohen, “Models of Practical Defenses Against Computer Viruses”, IFIP-TC11, “Computers and Security”, V7#6, December, 1988.
- 13 - M. Cohen, “A New Integrity Based Model for Limited Protection Against Computer Viruses”, Masters Thesis, The Pennsylvania State University, College Park, PA 1988.
- 14 - F. Cohen, “A Cryptographic Checksum for Integrity Protection”, IFIP-TC11 “Computers and Security”, V6#6 (Dec. 1987), pp 505-810.
- 15 - Y. Huang and F. Cohen, “Some Weak Points of One Fast Cryptographic Checksum Algorithm and its Improvement”, IFIP-TC11 “Computers and Security”, V8#1, February, 1989
- 16 - F. Cohen, “A Cost Analysis of Typical Computer Viruses and Defenses”, IFIP-TC11 “Computers and Security” 10(1991) pp239-250 (also appearing in 4th DPMA, IEEE, ACM Computer Virus and Security Conference, 1991)

- 17 - M. Pozzo and T. Gray, "An Approach to Containing Computer Viruses", Computers and Security V6#4, Aug. 1987, pp 321-331
- 18 - J. Page, "An Assured Pipeline Integrity Scheme for Virus Protection", 12th National computer Security conference, Oct. 1989, pp 369-377
- 19 - M. Bishop, "An Overview of Computer Viruses in a Research Environment", 4th DPMA, IEEE, ACM Computer Virus and Security Conference, 1991
- 20 - F. Cohen, "A Summary of Results on Computer Viruses and Defenses", 1990 NIST/DOD Conference on Computer Security.
- 21 - M. Harrison, W. Ruzzo, and J. Ullman, "Protection in Operating Systems", CACM V19#8, Aug 1976, pp461-471.
- 22 - F. Cohen, "A DOS Based POset Implementation", IFIP-SEC TC11 "Computers and Security" (accepted, awaiting publication, 1991)
- 23 - B. W. Lampson, "A note on the Confinement Problem", Communications of the ACM V16(10) pp613-615, Oct, 1973.
- 24 - D. E. Bell and L. J. LaPadula, "Secure Computer Systems: Mathematical Foundations and Model", The Mitre Corporation, 1973
- 25 - F. Cohen, "A Note On High Integrity PC Bootstrapping", IFIP-SEC "Computers and Security" (accepted, awaiting publication, 1991)
- 26 - M. Joseph and A. Avizienis "A Fault Tolerant Approach to Computer Viruses", Proceedings of the 1988 IEEE Symposium on Security and Privacy, 1989
- 27 - F. Cohen, "A Note on the use of Pattern Matching in Computer Virus Detection", Invited Paper, Computer Security Conference, London, England, Oct 11-13, 1989, also appearing in DPMA, IEEE, ACM Computer Virus Clinic, 1990.
- 28 - J. Herst, "Eliminator" (Users Manual), 1990, PC Security Ltd. London, ENGLAND
- 29 - F. Cohen, "The ASP Integrity Toolkit - Technical Support Center Manual", 1991 ASP Press, PO Box 81270, Pittsburgh, PA 15217, USA
- 30 - C. Shannon, "A Mathematical Theory of Communications", Bell Systems Technical Journal, 1948.
- 31 - C. Shannon, "A Communications Theory of Secrecy Systems", Bell Systems Technical Journal, 1949.
- 32 - D. Huffman, "A Method for the Construction of Minimum Redundancy Codes", Proceedings of the I.R.E. V40, pp1098-1101, Sept. 1952
- 33 - M. Breuer and A. Friedman, "Diagnosis and Reliable Design of Digital Systems", Computer Science Press, 1967 (see 'rollback')
- 34 - F. Cohen, "Defense-In-Depth Against Computer Viruses", Computers and Security (submitted, 1991).
- 35 - Special issue of the Communications of the ACM, V?#?, 1989

- 36 - F. Cohen, et. al., “A Collection of Short Essays on Information Protection”, 1990, ASP Press, PO Box 81270, Pitttsburgh, PA 15217, USA
- 37 - J. VonNeumann, “The Complete Works of John von Neuman’
- 38 - C. Langton, ed. “Artificial Life” 1989, Addison-Wesley Publishing Company, New York, NY.
- 39 - J Shoch and J Hupp, “The ’Worm’ Programs - Early Experience with a Distributed Computation”, CACM pp172-180, March, 1982.
- 40 - A. Dewdney, “Computer Recreations”, in Scientific American, 1984-1986
- 41 - for details, write to ‘The Computer Virus Contest’, ASP Press, PO Box 81270, Pittsburgh, PA 15217, USA.