Current Best Practice Against Computer Viruses With Examples from the DOS Operating System

by Dr. Frederick B. Cohen ‡

In this paper, we discuss the current best practice against computer viruses in practical systems. We begin with a summary of research on viruses and defenses. We then examine the state-of-the-art in virus defense today and describe how normal computing activities can proceed without undue risk of substantial viral harm. We then quickly summarize, draw conclusions, and suggest future work.

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

1 Background

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 [2], 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.

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 were severely limited by restrictions on research. [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 computer 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], by March of 1991, 600 real-world viruses were known in the research community [10], and as of October, 1991, the number had

reached 1,000 viruses with 30 new viruses appearing per week. [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. In the remainder of this paper we will look at how software based fault-tolerant computing techniques have been used to deal with the computer virus problem.

2 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
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]

3 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 valid. [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, 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 a 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.

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 and viruses 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]

4 Automated Repair

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, cus-

[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

tomized 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 a modification that is hard to reverse from the information in the infected file is the addition of innocuous instructions at random locations in the infected program. We would have to be able differentiate the 'legitimate' instructions from the 'illegitimate' ones in order to unweave the program. It is fairly easy to make this unweaving as hard to solve as the halting problem.

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). [29]

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. [29]

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.

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 by limiting information flow to 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. [25] BootLock mechanisms provide 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 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 known virus 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]

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. To give a clearer picture, we describe one operational system for a PC. [29] This system is implemented through a series of techniques installed and operable at different phases of DOS operation (see figure), where processing proceeds from phase p_1 to p_5 and then alternates between p_6 and p_7. Phase p_1 can only be altered by hardware changes, whereas all other phases of operation are easily modified by any DOS program under normal DOS operation.

Phase	Operations
p_1	Hardware + ROM bootstrap
p_2	Disk bootstrap
p_3	Operating system load
p_4	Driver loads
p_5	Command interpreter load
p_6	Instruction processing
p_7	Operating system calls

- Phase p_1 cannot be altered in software.
- Phase p_2 is modified by adding a BootLock to prevent external access and unauthorized hardware calls that modify the bootstrap procedure.
- Phase p_3 varies substantially from version to version of the DOS operating system, and cannot be reliably modified as far as we can determine. Therefore, no special provisions are included in this phase.
- Phase p_4 is modified to include a 'login' process, and depending on which protection mechanisms are active, optionally performs SnapShot generation or restoration [5], checks external checking mechanisms against internal stored values, uses the verified external checking mechanisms to check critical operating system areas and files used in p_2 thru p_7 and automatically restores them from on-line backups if appropriate, performs an initial scan for known viruses, and requests a user ID and password for access control. Assuming the SnapShot mechanism is active and operates properly, the machine state is set to the stored p_4 machine state, just before checking, thus eliminating all exposures other than modifications to the SnapShot mechanism and the stored memory image. Assuming a valid user ID and password are given, all relevant operating system calls are rerouted through the protection mechanism which is loaded into memory and remains resident from this point on.
- By phase p_5, all protection mechanisms are in place, and the verified command interpreter is loaded. If previous checking phases result in uncorrectable fatal errors, and assuming the fail-safe values are appropriately set, the system never reaches p_5, and bootstrapping fails. Otherwise, we are now operating in a high integrity state.
- In phase p_6, protection is not active, because there is no hardware to support memory protection or limited instruction sets, and thus no protection is feasible without simulating the entire operation of the computer, which would dramatically impact

[5] A new SnapShot is generated if a previous one did not exist, otherwise, the previous startup state is restored. Operation then continues from the PC startup image generated on prior bootstrap.

performance. It is then the task of the resident mechanism, if and when activated, to attempt to counter malicious actions taken in phase p_6.

- In phase p_7, operating system calls are intercepted by the resident protection mechanisms. Assuming that phase p_7 passes through the resident protection mechanism, it is broken into several subphases:

phase	Operations
$p_{7.1}$	trap bypasses
$p_{7.2}$	access control
$p_{7.3}$	execution checks
$p_{7.4}$	other traps

  - In phase $p_{7.1}$, mechanisms prevent automated attacks from bypassing protection. There is no way to guarantee that the applications program will not alter the resident protection mechanism or that it will enter that mechanism for p_7 operations, and there are several common methods for bypassing these mechanisms, including; bypassing the operating system altogether; tracing operating system calls as they operate to determine when the real DOS entry point is reached; and using known addresses of common DOS entry points or obscure or undocumented DOS entry points.
    - Bypassing the operating system altogether is typically only used to modify very standard portions of storage, since each version of DOS may use slightly different internal structures. Modifying most of these areas is prevented by BootLock protection installed in p_2. These areas are also tested for change and corrected with on-line backups during phase p_4.
    - Tracing can be stopped in most cases by a clever enough defender, although very few defenders have succeeded in doing this effectively against the large number of possible implementations.
    - Bypassing the resident mechanism with known DOS addresses or undocumented operating system calls can be avoided by modifying DOS operating system areas so that they fail unless called through the resident protection mechanisms. If properly called, DOS areas are temporarily repaired for the duration of authorized calls and then remodified after the call is processed.

    In each of these cases, to be effective against a serious attacker, the mechanisms must operate in a large class of ways which varies with each use. This 'evolutionary' approach increases the computational complexity of the attacker attempting to find standardized defensive patterns and bypass them.
  - In phase $p_{7.2}$, access controls determine the accessibility of information based on memory resident access control lists loaded in phase p_4 based on the user ID. Inaccessible files produce error returns.

– In phase $p_{7.3}$, calls that load programs for execution cause the loaded files to be checked against known viruses and then checked for changes via cryptographic checksums. If changes are detected, automation provides for recovery and rechecking which, if successful, results in continued operation after correction without any side effects except the delay caused by recovery from on-line backups. In this phase, previous access controls are inactivated so that integrity checking and automated repairs which require additional access are facilitated. Thus operating system calls performed from this phase are not subject to access controls, except those in phase $p_{7.4}$.

– In phase $p_{7.4}$, trapping mechanisms are used to limit the actions taken by programs. As an example, if active or in phase $p_{7.3}$, executable files cannot be opened for read/write access. Since phase $p_{7.3}$ does not require the capabilities trapped by $p_{7.4}$, this assures that a virus which has bypasses other protection mechanisms will have increased difficulty bypassing protection during checking operations. More generally, we can limit all $p_{7.3}$ operations not strictly required for operation, and thus dramatically reduce exposures.

The following table summarizes these descriptions by showing each phase and each protection mechanism's status in that phase. In this table, '+' indicates the facility is optionally active, and '-' indicates the facility is inactive.

Facility\Phase	1	2	3	4	5	6	7	7.1	7.2	7.3	7.4
BootLock	-	+	+	+	+	+	+	+	+	+	+
SnapShot	-	-	-	+	-	-	-	-	-	-	-
Scanner	-	-	-	+	-	-	-	-	-	-	-
System Test	-	-	-	+	-	-	-	-	-	-	-
Passwords	-	-	-	+	-	-	-	-	-	-	-
Access Controls	-	-	-	-	+	-	+	+	+	-	-
Self-defense	-	-	-	+	-	-	+	+	-	-	-
Virus Monitor	-	-	-	-	+	-	+	-	-	-	-
Integrity Shell	-	-	-	-	+	-	+	-	-	-	+
Virus Traps	-	-	-	-	+	-	+	+	+	+	+

7 Experimental and Real-World Experience

In the laboratory and in operational experience, numerous experimental and real-world viruses have been tested against the above described 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 long enough to study them and improve the defenses before breakdown is complete. This is a vital point because with such a mechanism, we are now in a proactive posture, where the defenders are not 'chasing' the attackers, but rather the attackers are 'chasing' the defenders.

Defense-in-depth also allows us to use many weak mechanisms to help build a very strong mechanism, and to trade performance without losing much in the way of strength. For example, a 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, [6] 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 90% of the reported attacks. [7] [10]

The integrity shell [12,13] 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 'simulation virus' [27], A.K.A. a 'stealth virus' [34]) or circumvent the cryptographic mechanism [12]. This covers over 99% of the current known viruses, and in less vulnerable operating systems might be adequate on its own.

Access control has the effect of limiting the scope of an attack by preventing modification of transitively non-writable files by the attack. Thus, regardless of the access controls in use, all SUPs of the attackers 'domain' in the partially ordered set (heretofore POset) formed under transitivity are vulnerable. [1,3,4] To avoid this protection 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 current 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. [20] Under an operating system which exploits hardware protection, bypassing this mechanism requires finding an implementation flaw or circumventing the normal system bootstrap process. Unfortunately,

[6] 4 or more new viruses per day as discussed above

[7] 100 out of 1000 known viruses currently represent over 90% of the attacks, so we get 90% coverage by testing for only 10% of known viruses.

the only true POset access control implementation to date is under DOS [22], and the other mechanisms which are closed under transitivity are not widely used, so transitive spread normally makes access controls ineffective.

The series of virus traps which prevent damage by a variety of means are over 99% effective, but a skilled attacker can eventually bypass these techniques. The traps are designed to cover the weaknesses of other defenses (e.g. one trap prevents tracing calls into the operating system which could be exploited to bypass the POset), so that they tend to be far more effective than they would be on their own.

The remapping of disk areas at bootup prevents over 99% of the current automated physical attacks and the vast majority of simple manual attacks other than those performed by a skilled and well tooled operator with physical access to the hardware. Ultimately however, the bootstrap mechanism must be able to access the disk, and the best we can realistically hope to do is have this access limited by a cryptographic key provided by the user or physical limits on access to the system.

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. None of the 1,000 viruses we are aware of have been successful at escaping this defense, s we could justifiably claim that the likelihood of bypassing it is less than 1 in 1,000 (i.e. over 99.9% effective), but as a matter of policy, we don't rate any defense as better than 99% effective.

A simple [8] calculation, assuming independence of mechanisms, is that the likelihood of successful attack is less than 1 in (90x99x99x99x99x99), or less than $1.2\text{x}10^{-12}$! Unfortunately, protection just doesn't work that way, because we don't have random attackers. A serious attacker will divide and conquer, bypassing each mechanism in turn, and then exploiting the bypassed mechanism to bypass the next defense. Thus it is more likely that a successful attacker will succeed by spending the required time to bypass each mechanism in turn. Indeed, an attacker with physical access, intimate knowledge of the defense, and good tools, can bypass the defenses on a single system in a relatively short time. The question from a standpoint of overall virus defense is not, however, the ability of such an attacker to bypass the defense, but rather, the ability of such an attacker to do so on a regular basis by writing an independent program.

For a defense to be successful against automated attack, it is merely necessary to drive the complexity of systematic attack up to a level where programs cannot succeed in a timely fashion. For any static defense, this doesn't work well because selective survival assures us that as soon as an attacker bypasses a mechanisms once, that attack will work again and again. The method we use to eliminate this problem is automated program evolution. [1] By designing defenses that evolve with each use, we can dramatically increase the complexity

[8] but misguided

of automated attack.

For example, bypassing access controls may be done by finding the decision point in memory where acceptable access is differentiated from unacceptable access, and disabling the decision in favor of always granting access. It takes some time for a person to do this, but once it is done once, it can be automated (e.g. by searching for the pattern of instructions leading up to the decicion point). Suppose however, that each time the access controls were loaded, they were evolved into a different form, so that there were no common sequences of bytes. With a general form of evolution, detection can be made undecidable. [1,4] In practice, it is easy to evolve programs so as to make their detection by pattern NP-complete.

Ultimately, the battle between virus and defense can be won on a PC at each system bootstrap by the program that runs first. As an example, using a simulation virus that simulates the entire machine can effectively defeat any programmed defense. Indeed, these mechanisms are only effective to a limited extent because they operate at the operating system level, and viruses can operate at any level where information is interpreted in a general manner. For example, the integrity shell will be ineffective against viruses that spread from database cell to database cell through the use of database macros, since there is no way to tell legitimate cell changes from illegitimate ones unless the interpretation mechanism is able to make the distinction. The POset will prevent these viruses from spreading beyond the transitive closure of information flow, and the other mechanisms will prevent such a virus from impacting executables, but still such a virus could survive. Since no such defense is or can be perfect for the PC, we will perhaps always struggle with the problem of viruses.

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 100 known viruses on a slow PC in under 1/2 second. If we expand to 1000 viruses, this time exceeds 5 seconds, 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 is only about 17Kbytes long [13], so integrity shell operation slows typical program startup by under 0.2 sec.

In some cases, programs are used so often that even a small overhead has a substantial impact on overall system performance. In these cases, other methods must be used to minimize the performance impact while still retaining a reasonable degree of protection. Similarly virus monitors tend to produce false positives, and this must be addressed in a reasnable way in order to maintain utility.

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.

8 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 to protection. 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.

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.

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 far more 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.

9 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 - The Virus Bulletin, 21 The Quadrant, Abingdon Science Park, Oxon, OX14-3Y3, England