An Approach to Containing Computer Viruses *

Maria M. Pozzo
Terence E. Gray
Computer Science Department, University of California, Los Angeles, Los Angeles, CA 90024, U.S.A.

This paper presents a mechanism for containing the spread of computer viruses by detecting at run-time whether or not an executable has been modified since its installation. The detection strategy uses encryption and is held to be better for virus containment than conventional computer security mechanisms which are based on the incorrect assumption that preventing modification of executables by unauthorized users is sufficient. Although this detection mechanism is most effective when all executables in a system are encrypted, a scheme is presented that shows the usefulness of the encryption approach when this is not the case. The detection approach is also better suited for use in untrusted computer systems. The protection of this mechanism in untrusted computing environments is addressed.

Keywords: Infection, Integrity, Trojan horse, Computer viruses

Terry Gray is Director of the UCLA Distributed Systems Laboratory, and an Adjunct Associate Professor in the UCLA Computer Science Department. He received a BSEE from Northrop University in 1967, and a Ph.D. in Computer Science from UCLA in 1978. Prior to joining the UCLA faculty, Dr. Gray was Manager of Software Engineering at Ampex Corporation. He has also served on the technical staff of Transaction Technology, Inc. and Bell Laboratories. Current research interests include distributed system architecture, computer security, software engineering, advanced office systems, and the social impact of technology. He is a member of IEEE, ACM, Tau Beta Pi, and Upsilon Pi Epsilon.

* This research was supported in part by the NSF Coordinated Experimental Research program under grant NSF/MCS 8121696 and by the IBM Corporation under contract D850915.

North-Holland
Computers & Security 6 (1987) 321–331

0. Introduction

The infection property of a malicious computer virus [1] which causes modifications to executables is a concern in computer security. Modifications to executables are much less noticeable than those made to text or data files, and often go undetected. Such modifications often cause unexpected, unauthorized, or malicious side-effects in a computer system. This discussion is primarily concerned with the infection property of a malicious computer virus which causes modifications to executables.

Protecting executables from modification can be accomplished in two general ways: (1) by rendering the executable immutable and thus preventing all modifications, or (2) by detecting any changes to the executable prior to its execution. The first method can be accomplished by storing the executable in a read-only storage medium such as a ROM, a read-only directory or a read-only disk. Thus, the protection mechanism is coupled with the system or storage medium employed; its usefulness relies upon the security of the underlying system. Even if complete confidence in the security of the operating system is warranted, there is a problem in employing discretionary access controls (DAC) for read-only protection. Current implementations of DAC are fundamentally flawed [2] in that programs executing on a user's behalf legitimately assume all the user's access rights. This flaw could allow a computer virus to perform modifications despite read-only protection. The second method (detection) can be accomplished

Maria Pozzo received her M.S. degree in Computer Science from the University of Connecticut in 1981. She began her work in the area of Computer Security in 1982 while working for the Mitre Corporation. She was a member of the Multics Evaluation team and subsequently worked for Honeywell Information Systems to help achieve a B2 security rating for Multics. She is currently working on her Ph.D. at UCLA pursuing research in computer security.

by encrypting the executable using conventional or public-key cryptography, or by recording a cryptographic checksum, so that any modification can be detected prior to execution. The detection approach links the protection mechanism to the object to be protected rather than the system or storage medium, and thus, its usefulness depends less on the security of the underlying system. Furthermore, modifications to executables can still be detected outside the realm of a particular system. For example, if encryption is used when transferring executables between sites, modifications can be detected at the destination site. This technique is promising for protecting software distribution.

This discussion is concerned with the integrity of executables and provides a generalized mechanism for detecting modification of executables and limiting the potential damage of a computer virus. This research was motivated by recent work in the area of computer viruses [1]. Section 1 provides background material on computer viruses and discusses the seriousness of the virus problem. Section 2 describes the encryption mechanism and proposes one possible implementation. This mechanism is then discussed by considering its strengths and weaknesses in Section 3. An overview of the current prototype and a discussion of the open issues is presented in the conclusion in Section 4. The issues discussed in this paper are part of an ongoing effort. Our long-range goal is to develop a complementary set of independent mechanisms for protection against computer viruses and other malcious programs.

1. Computer Virus Background

A malicious computer virus, like a Trojan Horse [1], lures unsuspecting users into executing it by pretending to be nothing more than a useful or interesting program [4], while in reality it contains additional functions intended to "...gain unauthorized access to the system or to [cause a] ...malicious side effect" [5]. Programs of this type are particulary insidious because they operate through legitimate access paths. The difference between a computer virus and a traditional Trojan Horse is that a virus "...can 'infect' other programs by modifying them to include, a possibly evolved, copy of itself" [1]. The process of infection is depicted in Fig. 1.

The victim's file space contains several "clean" executables to which the victim possesses modify access. The villain creates an executable that performs a function designed to entice unsuspecting victims to invoke it. Embedded in the executable is a piece of clandestine code that is a virus. When the program is executed, the hidden viral code is executed in addition to the program's normal service. The victim, however, only sees the normal service, and therefore, does not detect the presence of malicious activity. The virus program, when executed by the victim, carries the victim's access rights and, therefore, has modify access to all of the victim's executables as well as any other programs for which the victim has legitimate modify access. In this case, the virus spreads by directly copying itself to the target executables. Alternatively, the virus can spread by replacing the target executables with an executable that contains the virus.

Furthermore, when any other user with access to the victim's executables, invokes one of the infected programs, the virus spreads to that user's executables and so on. In addition to the spreading capability, the virus may contain other code, such as a Trojan Horse, intended to cause damage of some kind.

The most serious impact of a virus, however, is the rapidity with which it propagates through the system undetected. Worm programs (programs or computations that move around a network gathering needed resources and replicating as necessary) propagate with similar speed [6]. This potential widespread security problem is detailed in [1] and the potential damage to both the public and private sector is extreme.

Several properties of typical computer systems lead to an environment in which computer viruses can wreak havoc: the need for program sharing

[1] "The Trojan Horse works much like the original wooden statue that the Greeks presented at the walls of Troy – it is an attractive or innocent-looking structure (in this case, a program) that contains a hidden trick, a trick in the form of buried programming code that can give a hacker surreptitious entry to the system that unknowingly invites the Trojan Horse within its figurative walls. The Trojan Horse is very simple in theory, but also very effective when it works. The program that is written or modified to be a Trojan Horse is designed to achieve two major goals: first, it tries to look very innocent and tempting to run, and second, it has within itself a few high-security tasks to try [3].

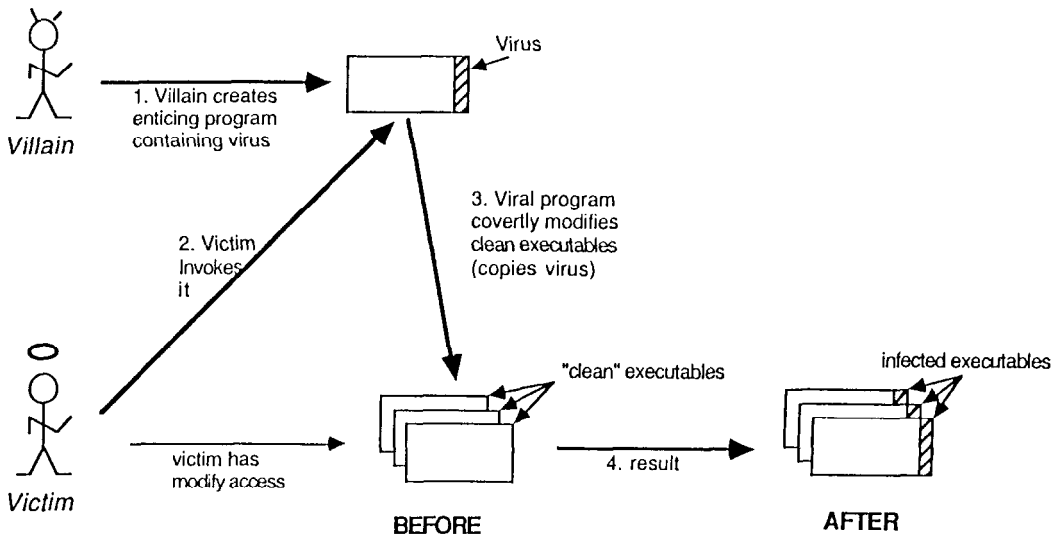

Fig. 1. The process of infection.

[5], the difficulty in confining programs [2], and the fact that existing discretionary access control (DAC) mechanisms are fundamentally flawed with respect to limiting computer virus [2]. Mechanisms exist for limiting the amount of sharing such as the security and integrity policies [8,9], and flow lists or flow distance policies [1]. However, to the extent that these mechanisms permit any sharing, the damage caused by computer viruses cannot be eliminated since their malicious activity is conducted via legitimate access paths due to the fundamental flaw in current DAC implementations.

1.1 Scope

Not all computer viruses are bad [1]. This discussion, however, is primarily concerned with the infection property of a malicious computer virus. Viral infection is caused by modification of executables. For this discussion, modification means directly changing the target executable or substituting the target executable with an executable that has been modified. Lastly, the system administrator discussed here is one or more persons trusted not to compromise the security or integrity of the system. This research does not address the case of a system administrator or other privileged systems user who has decided to corrupt the system.

[2] A program that cannot retain or leaks any of its proprietary information to a third party is confined [7].

2. Detecting Modification of Executables Using Encryption

Both conventional and public-key cryptography [10,11] have been used successfully to ensure the integrity of messages. Our solution proposes the use of cryptography to protect the integrity of executables, and thus provide a mechanism to detect viral spread and limit potential viral damage.

Fig. 2 depicts the proposed detection mechanism. The executable, E, is encrypted by the cryptosystem to produce E'. The run-time environment passes E' to the cryptosystem where the decryption is performed. The deciphered executable is passed back to the run-time environment which will attempt to run the results. If this attempt fails, the executable has been modified since it was encrypted and the proper authorities are modified. Thus, the run-time environment detects any modification, whether unintended or due to a viral attack. Note that modification of executables is not prevented; however, since any mod-

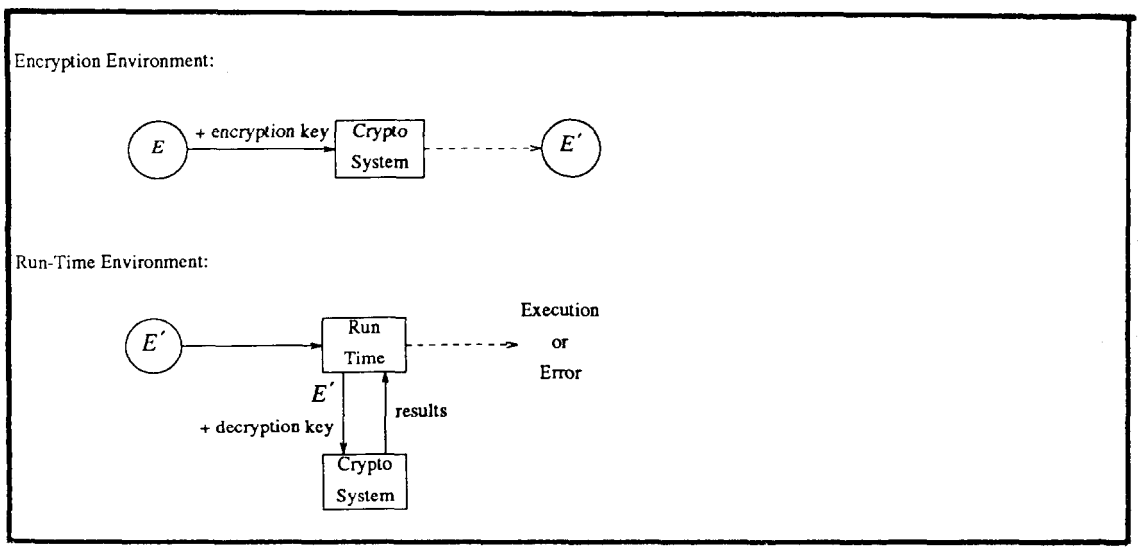

Fig. 2. The detection mechanism.

ification is detected at run-time (when the virus strikes), potential damage is limited. With respect to the damage that a virus can cause there are two cases to consider (Fig. 3). If all the executables in the system are virus-free and encrypted, attempts to infect an executable by inserting a virus will be detected and, thus, this mechanism completely halts any further viral damage. If a virus exists in an executable prior to its encryption, its presence will not be detected by this mechanism. This type of virus can still spread to other executables but the infection will be detected when the encrypted, infected program is executed. The original virus, however, can still accomplish its hidden function and, in effect, will behave like a traditional Trojan Horse. It should be noted that denial of service does occur since the executables will be destroyed when the infection attempts to spread.

The usefulness of this approach is dependent on the type of cryptosystem chosen, particularly the management of the encryption and decryption key(s). There are several advantages to using public-key cryptography as opposed to conventional cryptography. In a conventional cryptosystem, the keys used for enciphering and deciphering are either the same, or each key can be computed from the other [2,13]. Thus protecting the key(s), not only from modification but also from disclosure, becomes essential to the protection of the entire mechanism. In a public-key cryptosystem, enciphering and deciphering is accomplished via two keys, a private key and a public key [11,12,14]. In the mechanism described above, the private key is used to encrypt the executable while anyone with knowledge of the public key can decipher it. Protecting the private key from disclosure becomes the responsibility of the key's owner and is no longer part of the mechanism itself. Thus, protecting the integrity of the mechanism becomes a matter of protecting the algorithms and the public keys from modification. In a public-key cryptosystem, the private key is bound to a particular individual or group of individuals. This affords the additional advantage of authenticating the identity of the encryptor of an executable which provides additional assurance that the executable has not been replaced by an imposter program. This can be accomplished in a conventional cryptosystem if the encryptor and decryptor agree in advance on the key(s) to be used, however, the practicality of this approach is questionable. Except for the need for authentication, simply storing characteristic values of executables and protecting these values from modification, would be sufficient to achieve the goal of detecting changes to an executable.

One possible implementation of the mechanism described above is to employ a public-key crypto-

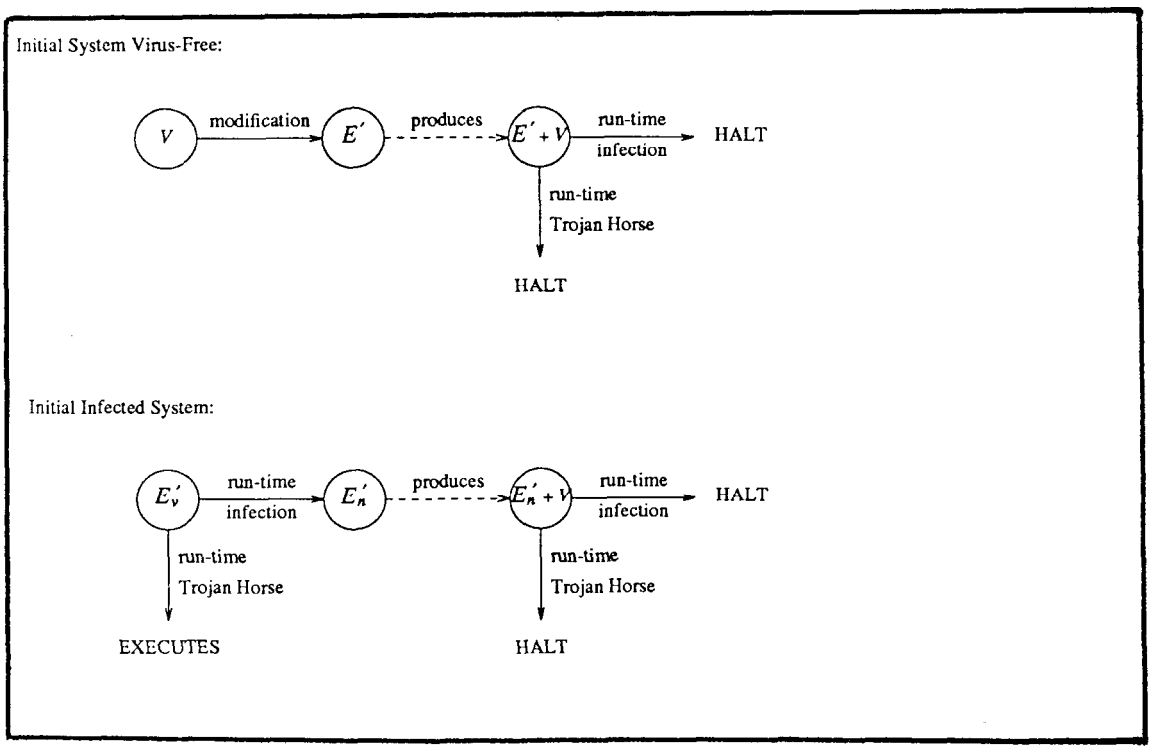

Fig. 3. Virus-free vs. infected system.

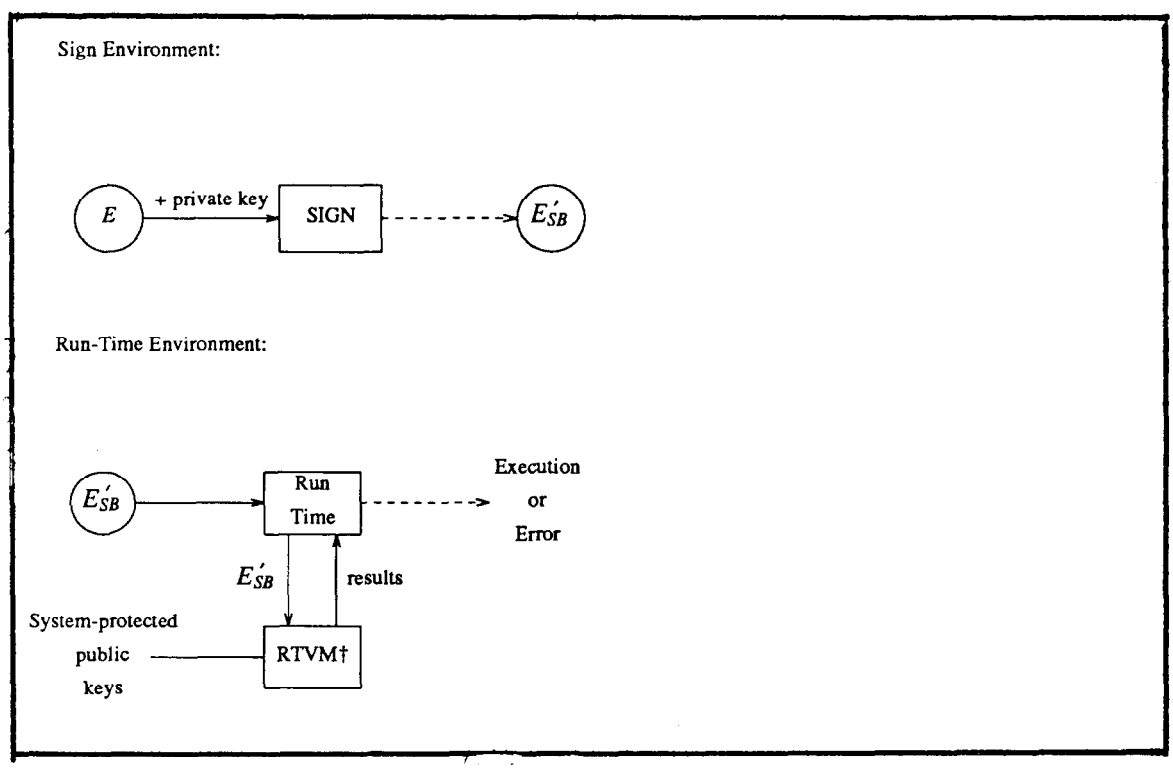

Fig. 4. Signature block mechanism. (RTVM = run-time validation mechanism).

system to append an encrypted signature block to the plaintext executable (Fig. 4). The signature block contains the result of applying a strong one-way function (characteristic value or cryptographic checksum) to the entire executable plus some additional information such as the identity of the signer and a time stamp. The private key of the signer is used to encrypt the signature block. At run-time, the signature block is deciphered with the associated public key, the characteristic value regenerated and the two values compared. If the two results differ, a modification has occurred and the proper authorities are notified. In this implementation, the system must maintain a list of public keys and protect them from modification. This implementation affords a large degree of flexibility in determining the set of public keys that will reside on the system-protected public key list (SPKL); these are the only keys that will be recognized by the run-time environment. In addition, since the executable itself is not encrypted, the run-time environment will not attempt to run deciphered code that is garbage. Authentication is a well-known advantage of signature block mechanisms [15], further indicating that this implementation is a viable solution.

Essential to the correct operation of this mechanism, is determining the public keys that reside on the SPKL. The system administrator should only allow public keys of individuals and organizations trusted to supply software that does not contain a computer virus. Basically, anyone can sign software; the issue is who can sign software and also have their public key on the public key list.

3. Strengths and Weaknesses

The mechanism described above is most effective if *all* the executables in the system are encrypted. In reality, however, it may not be feasible to require all executables in the system to be encrypted or signed. Of particular concern is the software development process which requires many executions during program debugging. Encrypting and deciphering on every test run will significantly lengthen this process. Providing a separate development environment, although one solution, may not be practical. Another concern is software developed by users for their own use, software not available to the entire system. Since residence in the system-protected public key list is restricted as described in the previous section, it is unlikely that the public key of a normal user will be in the list. This makes execution of personal software impossible. Lastly, the system administrator has the responsibility for ensuring that individuals and organizations who encrypt software for the system, and also have their associated public key on the system-protected public key list, provide software that does not contain computer viruses and other types of malicious programs such as Trojan horses. Requiring the system administrator to perform this task for all executables on the system may be impractical, depending on the degree of protection needed for the type of work performed by the system. Thus, for practical reasons, it may be necessary for encrypted and unencrypted executables to reside on a system simultaneously.

3.1 The Coexistence of Encrypted and Unencrypted Executables

One way to allow the coexistence of encrypted and unencrypted executables in a system is via the Risk Management Scheme. This scheme allows administrative classification of software, and permits users to specify the classes of software they wish to execute. The classes of software correspond to the likelihood that the executable contains malicious code such as a computer virus. Unencrypted software might be considered most likely to contain a virus. Thus, a user wishing to be protected from potential malicious activity would only allow execution of low-risk software. This classification, although subjective, serves as a warning mechanism to users, making them aware of the potential risk in executing certain programs. An overview of the Risk Management Scheme is presented here. For a detailed discussion see [16].

The Risk Management Scheme. The Risk Management Scheme has two domains: programs and processes. Programs are assigned "credibility values" (by the system administrator) and processes inherit a "risk level" from their parent process, or ultimately, from the user on whose behalf they are operating. The operating system is responsible for preventing a process at risk-level N from invoking a program whose credibility value is less than N.

The system administrator assigns software a

credibility value which identifies the likelihood that the software contains malicious code. In general, this value is based on the origin of the software. Credibility values range from zero to N, where software with the lowest credibility has the value of zero and software with the highest credibility on the system has the highest value. Software that is formally verified, so that the possibility of it containing malicious code is small, is always assigned the highest value. The number of credibility values is determined by the system administrator and can be one. For example, in an environment where security is of primary concern such as a military installation, a system may be restricted to only verified software. An environment where security is of less concern, is unlikely to have any formally verified software. But, since differences exist in the credibility of the various sources of executables, the system administrator can choose some number of credibility values to reflect the classes of software on the system. Fig. 5 depicts a possible configuration for credibility values.

Risk levels specify what classes of software can be executed for a user. They correspond inversely to credibility values. If the user's risk level is set to the highest credibility value on the system, the risk of damage to that user is the lowest possible. On the other hand, the greatest risk is taken when the user specifies a risk level of zero.

When a user logs in, a risk level is established for the session. This risk level can be determined in two ways. The first way is for the user to specify the desired risk level as an argument to the login command (e.g. login Joe-session_risk 3). The second way is to assume the default risk level for that user. Initially, the default risk level of all users is the highest credibility value on the system. The user can reset this default risk level by specifying the desired default as an argument to the login command (e.g. login Joe-default_risk 2). The user need only set this once and it remains in effect until it is explicitly reset by the user. Thus, assuming the default risk level as the risk level for the session requires no explicit action on the user's part once it is set. Once the risk level for a session is established, any processes that are spawned inherit the risk level of the parent, restricting children to running software of the same credibility value or higher. The only way for a user to override the risk level for a particular session is via the RUN-UNTRUSTED command which takes one executable program as an argument. This program can have a credibility value less than the risk level. The duration of this exception is the execution of the program supplied as an argument. The objective of the "RUN-UNTRUSTED" command is to make execution of high-risk programs explicit, but not too inconvenient.

As an example, Fig. 6 shows five possible credibility values for software, where the existence of malicious code in software with a value of 5 is unlikely and in software with a value of 0 is most likely. The initial default for the user is the ability to run software with a value of 5 only, unless the user explicitly logs in at a lower risk level or resets the default risk level. If the user chooses to establish a session with a risk level of 3, software with

Origin	Credibility	User's Risk
User Files	0 - Lowest	0 - Highest Risk
User Contributed S/W	1	.
S/W from Bulletin Board	2	.
S/W from System Staff	3	.
Commercial Application S/W	4	.
S/W from OS Vendor	5 - Highest	5 - Lowest Risk

Fig. 5. Credibility value and risk level.

Credibility Value	Execution Mode	User's Risk Level
0	RUN-UNTRUSTED	
1	RUN-UNTRUSTED	
2	RUN-UNTRUSTED	
3	normal	
4	normal	Risk Level = 3
5	normal	

Fig. 6. User's risk level.

values of 0, 1, and 2 cannot be run without using the RUN-UNTRUSTED command. Of course, the user has increased the potential risk of exposure to malicious activity.

Once a credibility value has been assigned to software, the information must be conveyed to the run-time environment. This can be accomplished in several ways. The first approach is to store the credibility value as part of the executable, comparing the value with the user's risk level prior to permitting execution. This approach requires that the executable be protected from modification to ensure the integrity of the credibility value. A second approach is to keep a list of all executable software in the system and the associated credibility values. When a user executes a program, the run-time environment searches the list for the program's credibility value and compares it with the user's risk level before allowing execution. Such a list must be protected from illicit modification. This approach may not be practical depending on the time it takes to complete the search. A third approach is to group software of the same credibility value in the same place in secondary storage, and maintain a short, protected list mapping credibility values to each file group. Software of the same credibility value could be stored in the same directory, in the same filesystem [3], or some other mechanism used to partition software. The list identifying each partition and the associated credibility value is then short enough to avoid performance problems, but must still be protected from modification by anyone except the system administrator. Fig. 7 shows possible credibility values for software grouped using Unix [4] directories as the partitions.

As the number of credibility values is determined by the system administration, so is the granularity of the partitions. For example, one system might partition all vendor software into one partition with the same credibility value while another system might have separate partitions for IBM, DEC and AT&T software, each with a different credibility value.

If an individual program becomes suspected of containing malicious code, perhaps based on reports from other installation, it can be moved to a different directory of appropriate credibility value. However, one disadvantage of associating a credibility value with entire directories or filesystems is that the full name of a program may be embedded in other programs or scripts; thus moving a program to a different directory having the desired credibility level is essentially a name change for that program, and may cause existing scripts to break. This observation argues in favor of assigning credibility values to individual programs, even though to do so is more administratively demanding. A combined approach that allows easy assignment of credibility levels to collections of pro-

[3] In Unix, a filesystem contains a hierarchical structure of directories and files and corresponds to a partition of a disk. Each filesystem is represented internally by a unique number [17].

[4] Unix is a trademark of AT&T Information Systems.

Origin	Credibility	Partition
User Files	0 - Lowest	/usr
User Contributed S/W	1	/usr/flakey
Bulletin Board S/W	1	/usr/net
Commercial S/W	2	/usr/bin2
S/W from System Staff	3	/usr/local
Commercial S/W	3	/usr/bin
Verified S/W	4	/usr/ver
S/W from OS Vendor	5 - Highest	/bin

Fig. 7. Partitioning software of different credibility values.

grams, but provides for individual exceptions may be the winning strategy.

Encryption Identification. Another major concern in allowing unencrypted and encrypted executables to coexist in a system is communication with the Run-Time Validation Mechanism (RTVM). There must be a way for the RTVM to know exactly which executables are required by the system administrator to be encrypted. Furthermore, this information must be trusted to accurately reflect the intention of the system administrator, i.e. it must be tamperproof. There are many ways to represent this "encryption identification"; several are listed below.

- Record the information as a protected attribute of the executable.
- Keep a system-protected list of all executables that are required by the system administrator to be encrypted.
- Group all encrypted and unencrypted software in the same place in secondary storage, and maintain a short, protected list identifying which locations must contain encrypted software.

In all cases, however, this information must be protected from illicit modification. For example, if an executable is identified as "must be encrypted" and this information is not protected from modification, a perpetrator could remove the encryption identification so that it is not validated by the RTVM. Essentially, unless the encryption identification is protected, the encryption mechanism is useless when unencrypted software is allowed to exist in the system.

3.2 Protecting the Protection Mechanism

The protection of the proposed mechanism itself is dependent on the integrity of the operating system. Protection of the mechanism does not require preventing disclosure of information, only its modification. Critical elements include the public key list (SPKL), and the RTVM. In systems where unencrypted executables are allowed, the encryption identification must also be protected as mentioned above. If the system is secure, the Trusted Computing Base (TCB) mediates all access between subjects and objects [18]. Routines that manipulate the public key list and the encryption identification would be considered privileged operations and part of the TCB. The operation of the RTVM would be considered a trusted operation and also part of the TCB. If the TCB provides multilevel security, additional protection is afforded by the security levels, since in general, a virus cannot spread between levels. [5]

If the underlying system is an Untrusted Computing Base (UCB), alternative measures must be taken to ensure the integrity of this mechanism. In addition to restricting valid public keys, the following issues are of primary concern:

- Protect the public key list and encryption identification from modification,
- Protect the routines that manipulate the public key list and the encryption identification from modification;

[5] The *-property [8] does not allow write-down to a lower security level.

- Limit execution of routines that manipulate the public key list and the encryption identification;
- Protect the Run-Time Validation Mechanism from modification.

For a more detailed discussion about protecting this mechanism when the underlying system is an Untrusted Computing Base see [19].

4. Conclusion

4.1 Open Issues

Performance issues are an area yet to be examined but an overall decrease in performance seems likely. This model requires the operating system environment to perform several additional services that will decrease performance. First, the credibility value of the software to be executed must be determined and compared to the risk level of the process executing the software. Secondly, the validation routines must be invoked for all encrypted software. The performance of the validation routine is dependent on the cryptosystem employed. Regardless of the one that is chosen, performance will be decreased.

Another open issue is that of name resolution. In the proposed model, when an executable is encountered with a credibility value lower than the process's default risk level, resolution is discontinued even if the entire name has not been examined. It may be possible to allow resolution to continue until an appropriate executable is found or the entire name has been resolved.

4.2 Current Prototype

The first prototype was implemented on the Locus distributed operating system [20], which is a network-transparent version of the Unix operating system. This prototype implements a framework for the signature mechanism. The primary goal of the implementation was to investigate the feasibility of protecting executables by using a signature block such as the one described. The Risk Management Scheme was not included in the first implementation.

A SIGN program was implemented in the initial system. The cryptosystem used, however, was trivial, and unsuitable for a real system. The partitions were simulated by using Unix directories. Both the SPKL and the list mapping the partitions into credibility values were assumed to already exist so that routines for manipulating them were not provided. The characteristic function was implemented by using the Unix "crypt" function to provide a 4-byte characteristic function. To test this mechanism, a program was written that invoked the RTVM if an executable resided in one of the "must be encrypted" partitions. If the executable was valid (not modified since it had been signed), it was executed.

Initial test results showed that modification to a signed executable would be detected in most cases. However, the characteristic function generator must provide a much stronger function than the 4-byte function supplied by the prototype. In general, however, any modifications made to the executable portion of the load module were detected. Appending a virus to the signed executable was also detected.

4.3 Future Work

The next step is to investigate a more rigorous characteristic function and to find a suitable public-key cryptosystem. The current simulation system must be moved to the operating system kernel and tested in real-time. A workstation may prove the best environment for the next level of the prototype. Once a more extensive signature mechanism is in place, the next step is to implement the Risk Management Scheme.

Once the entire model has been implemented, solutions must be found for the assumptions that were made. For example, a means for protecting the operating system kernel when the underlying system is an Untrusted Computing Base must be investigated. Also measurement of performance degradation introduced by the validation step is crucial to determining the overall feasibility of this approach.

References

[1] Cohen, F.: "Computer Viruses", *Proceedings of the 7th DOD/NBS Computer Security Conference*, September 1984, pp. 240–263.

[2] Boebert, W.E., and Ferguson, C.T.: *A Partial Solution to the Discretionary Trojan Horse Problem*, Honeywell Secure Technology Center, Minneapolis, MN.

[3] Landreth, B.: *Out of the Inner Circle: A Hacker's Guide to Computer Security*, Microsoft Press, Bellevue, WA, 1985.

[4] Anderson, J.P.: *Computer Security Technology Planning Study*, USAF Electronic Systems Division, Bedford, MA., Oct. 1972, ESD-TR-73-51.

[5] Denning, D.E.: *Cryptography and Data Security*, Addison-Wesley Publishing Co., Reading, Ma, 1982.

[6] Shoch, J.F., and Hupp, J.A.: The 'Worm' Programs – Early Experience with a Distributed Computation". *Communications of ACM* 25, 3 (March 1982), 172–180.

[7] Lampson, B.W.: "A Note on the Confinement Problem", *Communications of the ACM* 16 (10): 613–615, Oct, 1973.

[8] Bell, D.E., and LaPadula, L.J.: *Secure Computer System: Unified Exposition and Multics Interpretation*, MITRE Technical Report, MTR-2997, July 1975.

[9] Biba, K.J.: *Integrity Considerations for Secure Computer Systems*, MITRE Technical Report, MTR-3153, June 1975.

[10] Campbell, C.M.: The Design and Specification of Cryptographic Capabilities, *IEEE Communication Society Magazine*, Nov. 1978, pp. 273–278.

[11] Diffie, W., and Hellman, M.E.: Privacy and Authentication: An Introduction to Cryptography, *Proceedings of the IEEE*, Vol. 67, No. 3, March 1979.

[12] Meyer, C.H., Matyas, S.M.: *Cryptography: A New Dimension in Computer Data Security*, John Wiley & Sons, 1976.

[13] Kahn, D.: *The Codebreakers*, MacMillan, New York, 1972.

[14] Shannon, C.E.: "Communication Theory of Secrecy Systems", *Bell System Technical Journal*, 28, 1949, pp. 656–715.

[15] Kline, C.S., Popek, G.J., Thiel, G., and Walker, B.J.: "Digital Signatures: Principles and Implementations", *Journal of Tele-Communication Networks*, Vol. 2, No. 1, 1983, pp. 61–81.

[16] Pozzo, M.M., Gray, T.E.: "Managing Exposure to Potentially Malicious Programs", *Proceedings of the 9th National Computer Security Conference*, Sept. 1986.

[17] Bourne, S.E.: *The UNIX System*, International Computer Science Series. Addison-Wesley Publishing Company, 1983.

[18] DoD Computer Security Center, "Department of Defense Trusted Computer System Evaluation Criteria", DoD, CSC-STD-001-83, 1983.

[19] Pozzo, M.M., Gray, T.E.: "Computer Virus Containment in Untrusted Computing Environments", *IFIP/SEC Fourth International Conference and Exhibition on Computer Security*, December 1986.

[20] Walker, B.J., Popek, G.J., English, R., Kline, C.S., and Thiel, G.: "The Locus Distributed Operating System", *Proceedings of the Ninth ACM Symposium on Operating System Principles*, October 1983.