Defending Systems Against Viruses through Cryptographic Authentication

George I. Davida Yvo G. Desmedt Brian J. Matt

Electrical Engineering and Computer Science Department
University of Wisconsin-Milwaukee
Milwaukee, WI 53201

Abstract

This paper describes the use of cryptographic authentication for controlling computer viruses. The objective is to protect against viruses infecting software distributions, updates and programs stored or executed on a system. The authentication determines the source and integrity of an executable, relying on the source to produce virus free software. The scheme presented relies on a trusted (and verifiable where possible) device, the *authenticator*, used to authenticate and update programs and convert programs between the various formats. In addition, each user's machine uses a similar device to perform run-time checking.

1 Introduction

Computer security has been the subject of research for a number of years. The research in this area concerned itself with preventing unauthorized changes to programs and data [12, 14, 15, 18]. One type of attack that has been studied is the so called Trojan Horse attack. More recently the computer virus, a special type of Trojan Horse, namely one that can propagate itself, has heightened concerns about computer security both in the computer science community and the public [3]. Current computer systems offer limited protection against attacks by computer viruses. These viruses spread by working with the existing access control mechanisms infecting executable files. If left untreated, a virus can spread to the transitive closure of all system objects that are accessible by the users of the infected program(s) and are infectible. The success and rate of penetration are related to the design of the virus and such things as the number and the utility / popularity of the programs currently infected and their availability.

It is well known [1, 10, 19] that determining whether an executable has viral properties is hard. This is independent of whether the executable is stored on a disk, ROM, PLA or designed into a chip. However, determining if a virus has been *added* to an executable is simply detecting the modification of a file or message. Many techniques exist for detecting the modification of data. Error detecting codes[16] are widely known and are used for this purpose. However, while such codes may be sufficient for "random" and certain other types of errors, they are quite insufficient for protection against a determined adversary. Therefore, it is necessary that signatures be generated by strong cryptographic techniques.

Our view of software creation is similar to what one observes in the manufacturing of drugs. Just as people depend on pharmaceutical companies to produce uncontaminated products, users depend on software vendors' products being virus free. We assume that trustworthy software vendors have sufficiently clean environments that they can determine that their product, as released, is virus free. Our main goal is the protection of software from tampering during distribution and storage and execution. Consider software as if it were a drug. Our goal is to protect the product from the moment it leaves the "clean room" till it is consumed. Unlike a drug, software is consumed many times and must be protected even after it has been "injected" into the body, i.e. stored on disk in the user's system.

We assume that the software produced by a vendor is produced in a clean environment and is virus free. To protect software during distribution, storage and execution we:

1. Distribute software, releases and updates, to users in tamper proof "containers". The tamper proof nature of these devices is provided by cryptographic authentication.
2. Update software without introducing viruses into the new release.
3. Authenticate software, both when fetched from disk and during execution.

2 Previous Work

The concept of a virus has received considerable attention since it was first introduced, both in the academic literature and the popular press. The use of cryptographic verification to check programs prior to and during execution was proposed in Popek and Kline [18], Davida and Livesey [5] and Davida and Matt [6, 7, 8].

Cryptography, as a means of protection against viruses appeared in Pozzo and Grey [19, 20]. There a mechanism

CH2703-7/89/0000/0312$01.00 © 1989 IEEE

Petitioner Apple Inc. - Exhibit 1033, p. 1

was proposed to sign executable load modules using a public key cryptosystem [11], and check signatures as programs are loaded. Another approach suggested was to encrypt the entire executable. Cohen [4] presents another approach for checking executables as they are loaded.

A scheme based on linear feedback shift registers (LFSR) for protecting executables during run-time appears in Joseph and Avižienis [13]. Their approach involves having the compiler and loader generate a signed control flow graph (CFG) for a program using LFSRs and encrypting the result to form the program's "signature". At run time, the CFG is decrypted by a special device and the signatures checked as the program executes the instruction corresponding to the edges of the graph. Several open questions exist about their approach.

1. Flow control change instructions are not included in the signatures. Can a virus be installed by modifying only these instructions?

2. Is it possible to create a new version of an existing branch of a control flow graph that contains a virus? The signature generator used is LFSR based, and it is well known that LFSRs are vulnerable to known plaintext attack [2, 17]. The authors attempt to protect against this by encrypting the CFG and signatures. However, when an interrupt occurres, the contents of the LFSR must be saved along with the program counter in order to restart the LFSR after the interrupt has been serviced. Unless the interrupt stack is encrypted, or a separate physically secure stack is used, enough information will be available to determine the polynomial used. Unless the CFG is referenced at each interrupt return, the polynomial would have to be retained on a stack as well. The alternative to storing the LFSR at interrupt is to roll back to the beginning of the current CFG path on each interrupt.

   Once the program's polynomial is determined the signature for all edges of the CFG can be determined. An edge can be selected for infection such that the edge's signature, with the virus, is identical to the original.

3. Consider the following attack: **The Back Track Attack**

   (a) Dis-assemble the executable.

   (b) Add the virus to a load-module.

   (c) Assemble and link. The software generates new CFG with primitive polynomials and signatures and encrypts the new CFG. (Is this precluded by their remark about proper key management?)

   (d) The program is replaced with the new version.

The general form of the Back Track Attack is to back up from the executable, or come forward from source or load-modules, until the point where the encryption is performed. Then let program development tools (linker, assembler, etc) perform the encryption and replace the old version. All the designs mentioned [20, 19], and [13] appear to be susceptible to this attack.

The Back Track method of virus insertion is difficult to stop unless either programs in all their forms (source, load-modules, executables) are protected from their owners, or the users of signature generating programs are authenticated. The user or system manager[1] can supply a key for use in the program authentication process but the key must identify the individual as well, *i.e. not be supplied to the signature generating program automatically.* The act of generating a program signature must require the active user participation. This participation must authenticate the user, and verify that the user knowingly requests the program signature creation.

Having the authentication generator as part of an on-line system as has been done in previous works is dangerous. It is far easier to verify and protect a stand alone device.

3 Software Distribution

Our model of software distribution is the classical insecure communication channel [9], as in Figure 1. Vendors ship software, both full releases and updates, either directly to end users or to retailers. While undergoing development, or being processed by the authenticator, we assume that the software is protected. While in storage, at the vendor awaiting shipment or at the retailer, or in the system itself, it is in an insecure channel.

The scheme consists of performing the following:

1. The vendor generates signed software.

2. The user is able to verify the signed software.

3. The user installs (/ or customizes) the software, producing a local executable module.

4. The user creates a signed, block and overall, module.

5. During execution, the user's machine checks the executing programs using a built in hardware authenticator.

For stand-alone single-user systems, steps 2 thru 5 can be performed on a machine incorporating a hardware authenticator. In multi-user systems, on the other hand, steps 2 thru 4 have to be done either on a separate dedicated authenticator system or, if the hardware authenticator unit is part of the computer system, during a special single-user session.

Our choice of hardware authenticator devices is motivated by the following:

1. **The Vendor Level**
   Given the nature of the vendor's environment it is "possible" to authenticate a product using strictly software. However, confidence in such a procedure will not be high.

[1] In a multi-user system many of the procedures described in this paper would be performed by a system manager.

Petitioner Apple Inc. - Exhibit 1033, p. 2

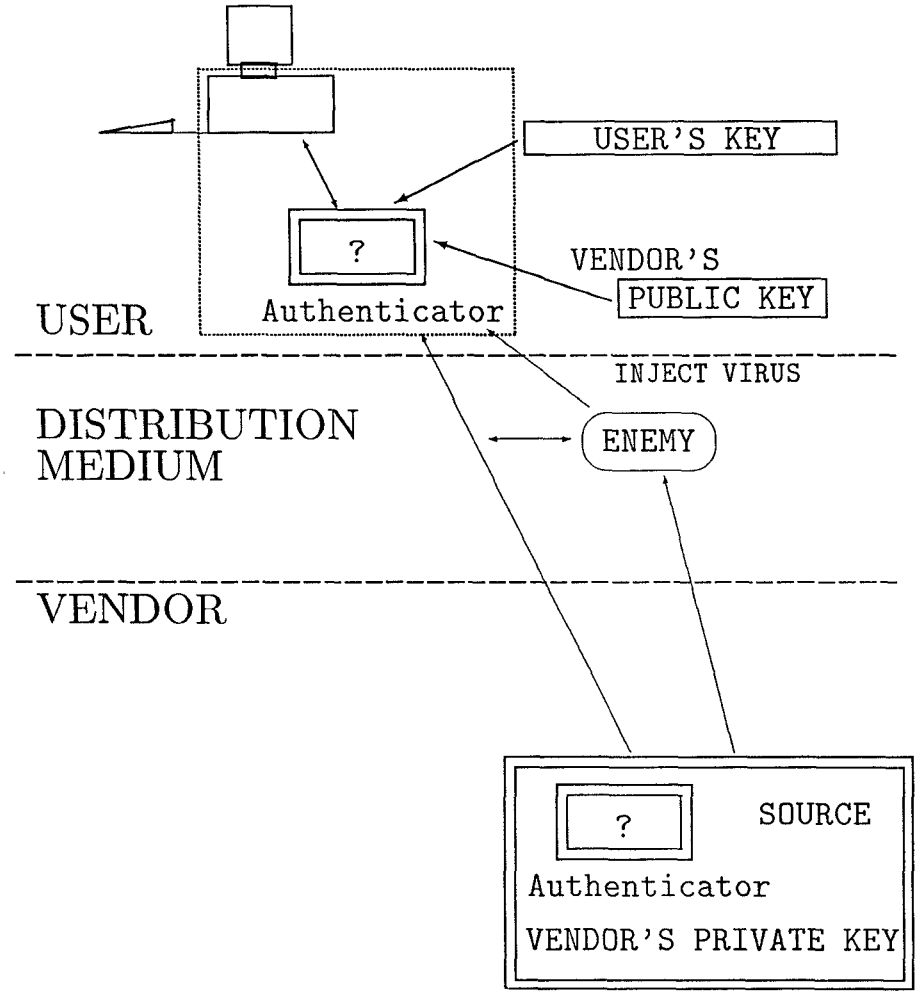

Figure 1: Software Distribution

2. **The User Level**
   At the user level it is even more important, if not absolutely necessary, to implement authentication in hardware. The users of a software product may be considerably less sophisticated than the vendor.

Hardware is in general more tamper resistant than software. Hardware authentication devices are less susceptible to software tampering than software authentication procedures.

A purely software authentication system must rely on considerably more complex protocols to verify the authenticity of a software product. A cold start must be employed and the normal system disk(s) must be removed. The disks/diskettes used in the authentication process are more vulnerable to tampering than a dedicated unit, and the initialization process is more complex than most ordinary users can be expected to handle.

Our solution utilizes public key cryptosystems for distribution of software between vendors and users; if you will, between a pharmaceutical house and the individual. Software retailers, like pharmacists and others with access to the product, are not trusted. The vendor signs the software distribution or update using a private key. The corresponding public key is available to all by means of a public directory. The user or system manager authenticates the software by using the vendor's public key.

Cryptosystems are used to authenticate software in the user's possession. The user must first authenticate the vendor's software and then "seal" the result by creating his/her own authenticator for it. For software systems that need to be configured (or customized) at the user site, this step is essential if the authentication of an *installed* software module is to be carried out without going through a *reinstallation*

Petitioner Apple Inc. - Exhibit 1033, p. 3

process and authentication of the original *unconfigured* product.

The cryptosystem used to generate the user's seal may be a conventional cryptosystem or a public key cryptosystem. If a public key system is chosen it may well be the same system used by the vendor.

To continue with the pharmaceutical analogy, the consumer takes the drug home and checks the "tamper proof" seal(s). If the seals are in an acceptable state, the drug is removed from the vendor's package and may then be further protected, as when parents keep drugs in child-proof cabinets or containers. The product may be placed in a medicine cabinet (software kept off-line) or "injected" immediately into the system (placed on-line).

The creation of signed software distributions by the vendor is performed by the vendor's authenticator device. The additional signing of the installed software is carried out by the user's authenticator.

The vendor's authenticator performs the following steps for a software release:

1. Read the vendor's private key.
2. Generate the signature for the release including the following pertinent data: program name, serial number, version number and date of release.

The user's authenticator performs the following steps for a software release:

1. Store the vendor's public key and the user's key.
2. Verify that the signature for the release is correct and display pertinent data: program name, serial number, version number, date of release, and so on.
3. Configure the program, if necessary.
4. Create a block and overall signed program.

It is the block-signed version that is loaded into the system.

The block signature scheme involves generating a signature for every page / segment (independently loaded block) along with a new overall signature. The block signature method utilizes the starting logical address of the block and a system wide program identification / version number, in the signature generation. The logical address is used to prevent any re-ordering of the blocks in the program. The use of the program identification / version number prevents substitution of signed blocks from another program or a different version of the same program[2].

The signed version is checked at run time by the authenticator unit located on the user's machine, see Section 5.

[2]As an alternative to program identification numbers, different keys could be used to sign each new version of each program but the number of keys becomes too large. Another approach is to use fewer keys and incorporate a program / version identifier into the key.

3.1 Updates

Performing an update requires the combination of the most recent complete release of the software available to the user, with one or more updates. Updates are distributed, as opposed to sending out new releases to reduce costs. If the amount of information necessary to patch the software is significantly less than would be contained in a new release, then an update is chosen.

To perform an update the vendor's authenticator creates a signed update by performing the following steps:

1. Read the vendor's private key.
2. Generate a signature for the new complete release, including the following pertinent data: program name, serial number, version number and date of release.
3. Generate a signature for the *updates* only including the following pertinent data: program name, serial number, version number and date of release.
4. Send the signed update for the new complete release to the users. In addition, the signature for the new complete release is sent. (The complete release is not sent.)

The user combines the update with the most recent complete release (see Figure 2) as follows:

1. Verify the vendor supplied update signature.
2. Perform the update creating the new release.
3. Verify the new release signature.
4. Create a new block and overall signed executable program.

4 Authenticator Design

Ideally, the authenticator is so simple and its users sufficiently sophisticated that they can verify the implementation of the authenticator. As a practical matter, the supplier of the authenticator or some "independent" testing laboratory would perform the verification.

How the design is implemented, the technology used, is critical not only to verification but to protecting the device against "hardware" viruses [10] as well. Viruses can be implanted into ROM chips, PLAs or coded into complex chips during the design process. Simple components make reverse engineering feasible and helps the detection of tampering. The use of simple components offers other advantages:

1. The complexity of the chips will be low.
2. Stock items can be used, making it more difficult for a manufacturer or supplier to infect chips destined for the authenticator.

Petitioner Apple Inc. - Exhibit 1033, p. 4

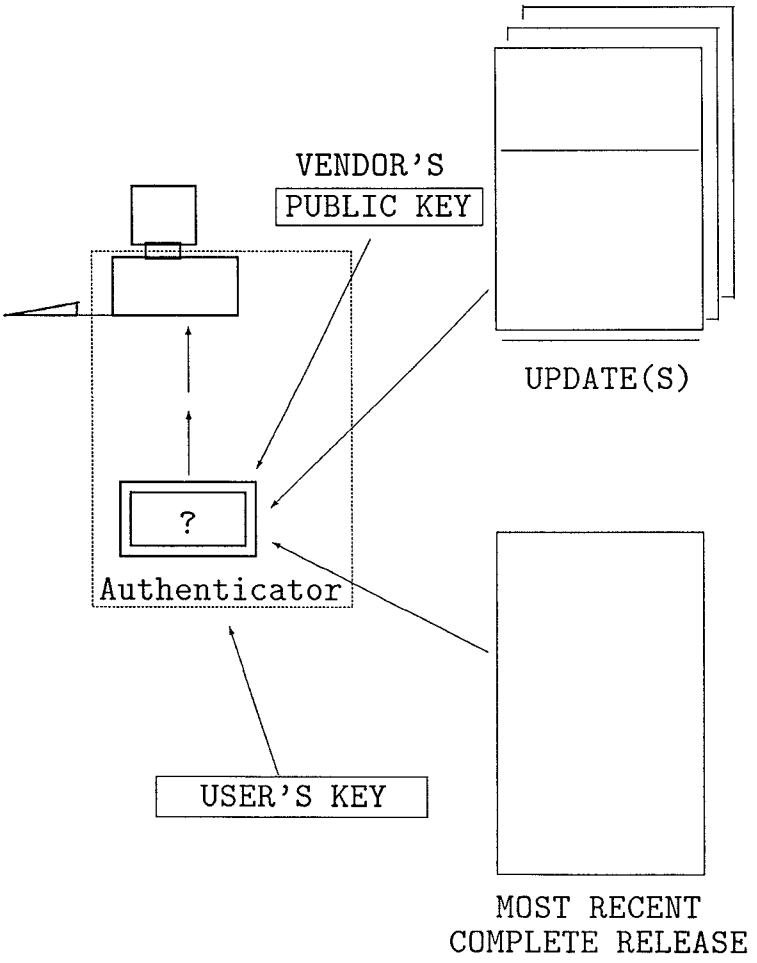

Figure 2: Software Updates by a User

3. Multiple chips should be necessary to perform a single function, for example a register. This is intended to force the virus to be spread across devices. If the chips used to perform the function are from different manufacturers the difficulty of implanting a virus is increased.

4. Older chips can be utilized. Items can be selected that were manufactured years apart. More significantly, chips can be used that are older than the virus concept.

When a user operates an authenticator, he/she along with the authenticator device provides a guarantee of the authenticity and integrity of the software. The authenticator creates a signed version that could include the vendor's signature.

This provides software users with more security than what is often available with drugs. Many drugs are distributed to pharmacists in large quantities, and though the container may have a seal when it arrives at the pharmacist, the seal is broken for the first customer and it is the pharmacist's responsibility to protect it from that point onward. Signing the result provides even more protection than locking the unsealed container in a safe.

Individual users may execute their own un-authenticated programs. In multi-user systems, users who wish to protect their own private programs against viruses may ask a system manager to have their programs signed. For stand-alone systems, this is obviously simple to do.

5 "Invivo" Protection

Even after the vendor's signature has been validated software is still vulnerable. The threat is not lessened by storing it in the user's machine. Since we consider the user's system to be a hostile environment, we must authenticate the code

Petitioner Apple Inc. - Exhibit 1033, p. 5

that we run as often as feasible. This is the function of the machine's authenticator.

The key stored in the authentication unit must be protected, both from (read[3]) and write access. In our design the (secret[3]) key is stored in the authentication unit directly. Even the operating system kernel, while using the unit to authenticate programs, does not have access to the key. The authentication unit must have its own external input/output channel for initialization.

Two main difficulties must be surmounted. Making sure that the operating system utilizes the authentication unit properly, and how often and how much of a program should be checked. We note in passing that the operating system itself may be checked as described above. This may even be done using "surprise inspections" techniques using externally generated schedules. Special architectural support is needed.

5.1 When to Check

A program can be checked in is entirety when it is loaded into the system and in addition, periodically by some background task. A complete check when the program is executed may not be feasible. Some systems use a demand load format which results in pages or segments being loaded from the executable only as needed. If, on the other hand, the authenticator unit is sufficiently fast and all of the program is brought into memory,(possibly with some of it being paged out into the page / spool area on disk), then it is feasible to check the entire program before executing it.

Even a running program (a process) is at risk. Whenever a process does not have control of the processor, it is vulnerable to attack[4]. Pages, segments, entire programs are moved by the operating system to and from the paging / swapping area on disk. While in the paging / swapping area this information is vulnerable to tampering. To protect against this, all data moving into main memory from the paging / swapping area, should be authenticated requiring appropriate architectural support.

The authentication may result in adjustments to the operating system's memory manager. If the cost of moving information between disk and main memory is significant, it may be necessary to decrease the number of concurrently executing programs or keep processes swapped out longer.

The importance of providing this level of protection should not be underestimated. Certain operating systems, (for example UNIX systems), allow for a program to be loaded into the page / swap area on first execution and have the text portion remain there, even after all current executions of it have finished. This copy is the "source" for all subsequent executions. Programs of this type are read, at most, once from the copy in the file system between system boots.

In addition to attacks mounted against the process by using the page / swap area, it is possible to infect the process by changing the information currently in memory. Whenever a context switch or interrupt occurs another program runs and all suspended programs are at some risk. This can be detected if instructions can be checked concurrently with their execution.

[3]If a conventional system is used.

[4]In the following we assume a uni-processor system.

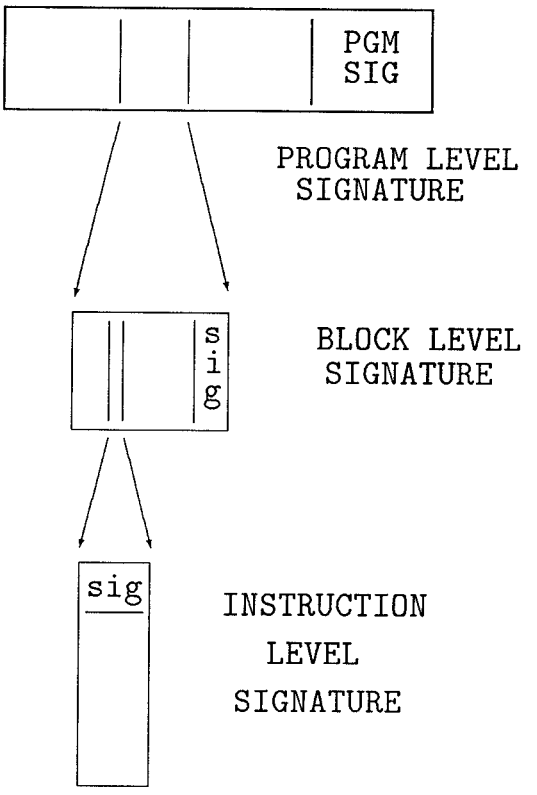

Figure 3: Signature Granularity

5.2 What to Check: The Granularity Problem

In the previous section we proposed that the authenticator signs the entire executable and units of pages and / or segments. We now propose that individual instructions can be authenticated as the CPU executes them. This authentication can be performed as part of the flow of control as in [13], but with all instructions signed, as in Figure 3. Instruction checking is performed in parallel with the execution cycle and generates a signature fault if it fails. The signature mechanism must prohibit relocation.

6 Remarks

The advantages of the proposed system are:

1. Increased confidence in software security.

Petitioner Apple Inc. - Exhibit 1033, p. 6

2. Simple checking mechanisms based on hardware which are more tamper resistant.

3. Updates are easily allowed.

4. Allows for variable granularity of checking during execution.

5. Separation of checking hardware from the system hardware.

7 Acknowledgments

The authors would like to thank the reviewers of this paper for several helpful comments.

References

[1] L. Adleman. An abstract theory of computer viruses. Presented at Crypto'88, Santa Barbara, California, U.S.A., to appear in: Advances in Cryptology. Proc. of Crypto'88 (Lecture Notes in Computer Science), Springer–Verlag, August 1988.

[2] A. Bauval. Cryptanalysis of pseudo-random number sequences generated by a linear congruential recurrence of given order. In *Eurocrypt 86*, pages 2.5A–2.5D, 1986. ISBN 91-7870-077-9.

[3] F. Cohen. Computer viruses: Theory and experiments. *Seventh National Computer Security Conference*, pages 240–263, September 1984. Also appears in Second IFIP International Conference on Computer Security.

[4] F. Cohen. A cryptographic checksum for integrity protection. *Computers and Security*, 6(6), December 1987.

[5] G. I. Davida and J. Livesey. The design of secure CPU-multiplexed computer systems: The master/slave architecture. *Proceeding of the 1981 IEEE Symposium on Security and Privacy*, April 1981.

[6] G. I. Davida and B. J. Matt. Crypto-secure operating systems. *National Computer Conference, 1985*, 54:575–581, 1985.

[7] G. I. Davida and B. J. Matt. UNIX guardians: Active user intervention in data protection. *Fourth Aerospace Computer Security Applications Conference*, December 1988.

[8] G. I. Davida and B. J. Matt. UNIX guardians: Delegating security to the user. *USENIX UNIX Security Workshop*, pages 14–23, August 1988.

[9] D. E. Denning. *Cryptography and Data Security.* Addison-Wesley, Reading Mass., 1981.

[10] Y. Desmedt. Is there an ultimate use of cryptography? In A. Odlyzko, editor, *Advances in Cryptology, Proc. of Crypto'86 (Lecture Notes in Computer Science 263)*, pages 459–463. Springer–Verlag, 1987. Santa Barbara, California, U.S.A., August 11–15.

[11] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. on Inform. Theory*, 22(6):644–654, 1976.

[12] V. D. Gligor, E. L. Burch, C. S. Chandersekaran, L. J. Dotterer, M. S. Hecht, W. D. Jiang, G. L. Luckenbaugh, and N. Vasudevan. On the design and the implementation of secure Xenix workstations. *Proceeding of the 1986 IEEE Symposium on Security and Privacy*, April 1986.

[13] M. K. Joseph and A. Avižienis. A fault tolerance approach to computer viruses. *Proceeding of the 1988 IEEE Symposium on Security and Privacy*, April 1988.

[14] P. A. Karger. Limiting the damage potential of discretionary trojan horses. *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 32–37, April 1987.

[15] T. F. Lunt and R. Jagannathan. A prototype real-time intrusion-detection model. *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 59–66, April 1988.

[16] W. W. Peterson and E. J. Weldon Jr. *Error Correcting Codes.* MIT Press, Cambridge Mass., 1972.

[17] J. B. Plumstead. Inferring a sequence generated by linear recurrence. *Proceedings 23rd IEEE Symposium on Foundations of Computer Science*, pages 153–159, 1982.

[18] G. J. Popek, M. Kampe, C. S. Kline, A. Stoughton, M. Urban, and E. J. Walton. UCLA secure UNIX. *National Computer Conference, 1979*, 48:355–364, 1979.

[19] M. M. Pozzo and T. E. Gray. A model for the containment of computer viruses. *Second Aerospace Computer Security Conference Applications Conference*, pages 11–18, December 1986.

[20] M. M. Pozzo and T. E. Gray. An approach to containing computer viruses. *Computers and Security*, 6(4):321–331, August 1987.

Petitioner Apple Inc. - Exhibit 1033, p. 7