Construction, Testing and Use of Checksum Algorithms for Computer Virus Detection

by

Douglas William Varney

B.S., University of Virginia, 1980
M.B.A., University of Virginia, 1984

A Thesis

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

in

Department of Computer and Information Sciences

Kansas State University
Manhattan, Kansas

1989

Approved by:

Major Professor

Table of Contents

Chapter 1 Introduction

1. Overview

As computers become more integral to daily lives, the integrity of the computer activities becomes increasingly crucial. To that end there has been increased research into the area of maintaining integrity in a computer environment.

The definition of integrity from the NIST Workshop on Integrity, January 1989 is:

> The property that data, an information process, computer equipment, and/or software, people, etc., or any collection of these entities, meet an a priori expectation of quality that is satisfactory and adequate in some specific circumstance.

The workshop and its related activities were held because there is a growing concern for integrity of information stored in computers and in machine readable format on storage devices. The threats to this integrity are numerous and serious ranging from those that can be considered unintentional to those that can be categorized as malice with forethought.

Most of the problems with integrity of data and programs can be categorized as unintentional. The entering of incorrect data is probably the largest threat to computer integrity, followed in a close second place by errors caused unintentionally. There is a growing concern about threats to integrity from programs whose actions intentionally do not meet their program's specifications, known as Trojan Horses. A Trojan horse is a piece of code that is surreptitiously placed in a program in order to perform functions not advertised by the program specifications. [MAE87] A type of Trojan Horse that is particularly dangerous is a computer virus. In this research a virus is defined as a program that can "infect" other programs by modifying them to include a, possibly evolved, copy of itself. With the infection property, a virus can spread throughout a computer system or network using the authorizations of every user using it to infect their programs. Every program that is infected may also act as a virus and thus the infection spreads [COH84]. Because viruses can spread so rapidly and have the potential to destroy the integrity of large amounts of data an effective means must be found to counteract this threat to integrity. This thesis describes research which copes with the problem of detection of virus infected files using techniques developed to maintain the integrity of files of data.

1.2 Models of integrity

Several models for insuring the integrity of computerized files have been advanced. Although the Biba model [BIB77] was introduced twelve years ago, the past three years has shown an increased interest in the integrity area. The recently introduced Clark and Wilson model [CLA87] [WIL89] has drawn particular interest. These two models, Biba and Clark & Wilson are described as follows.

1.2.1 Biba

The Biba Model is based on the definition of integrity as a multivalued quantity, versus the binary property of the integrity definition of the NIST workshop. With the Biba model, data and processes are given an integrity label in a range defined for the system. An integrity lattice for the system can be constructed from the integrity labels on the data items. If an implementation of the Biba model meets the model's specification, that implementation insures that a process can not reduce the integrity label and thus the integrity of a file of data.

The strict Biba model is a dual of the Bell-LaPadula lattice security model. [HEN87] A process in the Biba model is not allowed to write to data which have a higher integrity label (corresponding to "no write down" property of Bell-LaPadula) and is not allowed to read from data which have a lower integrity label (corresponding to the "no read up" property of Bell-LaPadula). Thus, information in a lower integrity level cannot corrupt information in a higher integrity level. Using a proof by induction, if data is in a valid integrity state it can be shown that at all future times it will remain in a valid state assuming a Biba model of integrity is imposed.

Biba also proposed two variants of his strict integrity policy, ring policy integrity and low water mark integrity. With the ring integrity policy no restrictions are placed on the reading of data, but the constraints on writing to an object are the same as the strict integrity policy. Low water mark integrity changes the subject's integrity label to that of the object's integrity label when the object's integrity label is less than that of the subject's integrity label. [HEN87] There are also other variations of the strict Biba integrity policy. [DEN86] [SHI81] [BOY78]

There are several problems with the Biba model and its variants. Strict Biba does not appear flexible enough to be useful in practical applications since these applications must have read and write access to various system tables and internal data structures in order to perform their functions. [HEN87] Another serious problem with a strict Biba integrity policy occurs when it is combined with the Bell-LaPadula security model. This combination causes isolation of data at lattice nodes to occur (this combination partitions systems into closed subsets under transitivity). [COH84] Strict Biba also has no automatic mechanism to incorporate new data into the hierarchy. When flexibility is introduced to counter the constraints of the strict Biba model there is migration of data to a lower level, as occurs with the low water mark integrity, or there is the problem of integrity corrupting mechanisms migrating across integrity levels. [COH84] Managing a Biba type implementation is also difficult. Most lattice model (Biba) designs to date have considered 64 categories to be a large number. [KAR88] Large systems will have thousands of distinct categories because to effectively limit the operations between a subject and similar objects that must be treated differently will require a separate label. Managing large numbers of categories is not unique with Biba systems and will extract performance penalties on all general integrity policies.

1.2.2 Clark & Wilson

The Clark & Wilson model insures the expectation that the integrity of systems and data remain predictably constant and change only in highly controlled and structured ways. Though the original Clark & Wilson paper [CLA87] was expressed in terms of nine rules, Lee captured the essence as:

> All data (of interest) must be modified by, and only by, authorized well-formed transactions where the principle of separation of duties is used to limit who can perform what transactions and make what changes to the system. [LEE88]

With the Clark & Wilson model internal consistency and good correspondence to real-world expectations for systems and data are provided. [WIL89] Correspondence to real-world expectations is accomplished by Integrity Verification Procedures (IVPs). These procedures check the model formed by data in the computer system against the real world perception of the model. The IVP not only provide correspondence to the real-world but also checks the internal consistency of the data. After an IVP the data has integrity. An example of a practical IVP is physically counting the inventory at a location and checking that the computer system designed for tracking that inventory corresponds to what was physically found.

A crucial second feature of the Clark and Wilson model is controlling change. Between IVP execution on a set of data any changes to the data must be strictly controlled in order to maintain internal consistency and thus integrity.

Controlling changes can take four forms determined by the structure and use of the data: prevention of change, attribution of change, constraint of change, and partition of change.

For data that does not change in the real world the **prevention of change** is desirable. Using the Clark & Wilson model, if it can be shown that the data was correct at one time and has not been changed then the integrity of the data is maintained. An example of a file where the use of prevention of change is appropriate would be a file of executable programs that rarely change.

For unstructured data the integrity of data can be determined if the data and author (original and of changes) are bound in an unforgeable way. If the data has been changed the integrity can be maintained by binding the history of the changes and the authors of those changes to the data. An example of data appropriate for the control mechanism of **attribution of change** would be memos or reports.

Highly structured data, such as accounting records, should only be modified in very controlled manners. If only certain programs and users are allowed to modify the data, this method is called **constraint of change.**

In order to prevent fraud, the changing of some types of data should require that the change be authorized by two different people, i.e. **partition of change.** Money transfers by wire should be controlled by this separation of duty.

1.3 Prevention of Change

This section will elaborate on the concepts involved in the prevention of change as it is the detection of change that we wish to focus upon. To prevent change, the system must either prohibit change through access control or identify that change has occurred and take appropriate action.

1.3.1 Access Control

It is possible to design a system in which there is a category of data that should not be changed. The prevention of modification is accomplished by some form of an access matrix model. The access matrix model consists of a triple (Subject, Object, Access Matrix). Subjects are active entities, Objects are protected entities to which access must be controlled, and the Access Matrix is a matrix in which rows correspond to subjects and columns correspond to objects, where a entry stores the access rights of the subject to the object. [MIZ87] Rights are the operations that the subject can perform on the object. Since the matrix tends to be very sparse (i.e. most subject - object pairs have no rights) the matrix typically is implemented as a list of subjects that have access rights to an object (Access Control List) or as a list of objects to which a subject has rights (Capability Lists). The two methods yield major differences in the type of protection provided.

Access Control Lists (ACLs) are the most common form of integrity (and security) control. It is a column-based view of the Access Matrix derived from the nonempty entries of an object. An object has a list of pairs (subject, right) indicating the subjects that have access to the object and the rights for each subject. Rights typically are read, write, and execute. If a subject, s, tries to access an object, the list of access (access control list) for that object is searched. If an entry for subject, s, does not exist or if it does exist but the requested rights do not occur in that entry the request is refused. Typically a subject acting on the request of another subject obtains the rights of the originating subject. For example, a user can execute a compiler which then will have all the rights of the user.

ACLs suffer problems in regards to integrity in both implementation and theory. The implementation is typically very coarse-grained in the size of objects and the small number of rights that can be granted. ACLs normally are applied at the file level, so they cannot maintain integrity for a part of a file that needs to be treated differently for access purposes. This is compounded by the small number of rights that are used. The combination of Read and Write are sufficient to accomplish all features of a computer system, but if only these are used (or even with the addition of execute) then the user may not be sure that data is modified in a manner maintaining integrity. The problem with implementation is not one of ACL theory. It should be possible to decrease the size of objects which are protected and increase the number of rights available but at an increase in the cost of storage and efficiency.

The theory of the ability to transfer rights is a much more serious flaw with respect to integrity. Programs operating on a user's behalf have all the rights of the user. Any data that is accessible for change by the user is accessible for change by the program executed for the user. This accessibility makes ACLs very vulnerable to any program that per-forms a surreptitious or unadvertised function, i.e. a Trojan Horse. If a Trojan Horse resides in the C compiler it then has the access rights to all the files to which the the user has access rights. Thus it can modify or delete any objects to which the user has write access.

The alternative form of the Access Matrix viewed from a row basis is the Capability List. A subject has a list of objects it has capabilities (rights) to which defines the domain of the subject. [MIZ87] When an object is invoked the system determines if it is in the capability list of the subject and, if so, allows the operation to continue. The implementation of Hydra [COH75] allows rights amplification which handles abstract data types easily. A good implementation of Capability Lists provides an excellent means of integrity control because it naturally provides a mechanism for each program to be executed in the smallest possible domain. [MIZ87] Due to other considerations such as the concept of ownership there are very few systems using Capability Lists.

There have been attempts to provide the integrity protection of Capability Lists without the drawbacks of Capability Lists. Two examples are the Four-tuple ACL [MIZ87] and the Access Control Triple [WIL89]. With the Four-tuple ACL, each subject in an ACL entry is represented by a four-tuple of user ID, class ID, module ID, and exported procedure name. This effectively limits the domain available for Trojan Horses to the same degree as the Hydra system. It also provides control over users because users can only view or change data through the levels of the subject IDs. A simpler concept is the Access Control Triple which binds user, program and data. Flexibility would not be as great as in a Four-tuple ACL since fewer grouping are possible, but implementation would be easier

1.3.2 Checksum Techniques

Another method of insuring that data has not changed is to attach additional information that at some level of confidence assures that the data has not changed. This is typically in the form of a checksum. A checksum, or digital signature, is any fixed length block functionally dependent on every bit of the message, so that different messages will have different checksums with a high probability. [DEN82] A checksum can be evaluated on two features: the ability to prevent forgery and the computational complexity of the algorithm that creates it. Checksums can be determined in two basic manners: using cryptography or using a deterministic (noncryptographic) algorithm.

Cryptography is defined as the methods and process of transforming an intelligible message into an unintelligible form and reconverting the unintelligible form into the original message through a reversal of the process of transformation. The original message is referred to as the plain text and the enciphered message is called the cipher text. A cipher system consists of the following two items: 1. A set of rules that comprise the basic cryptographic process (called the general system, is agreed upon in advance, and is constant in nature), and 2. A key, which may be variable. [KAT73]

Converting plain text to cipher text is known as encryption, while converting cipher text back to plain text is known as decryption. The process can be described by the transformation: plaintext —> cipher text —> plaintext or in other terms: f(plain text, encryption key) = cipher text; f(cipher text, decryption key)=plaintext. If the encryption key is not equal to the decryption key the cryptographic system is known as a **public key** cryptography system. If the encryption key is the same as the decryption key then the cryptographic system is known as a **private key** cryptography system. When the keys are different it is possible to broadcast or distribute (make public) the encryption key for other parties to send messages that only the parties knowing the decryption key can convert back to plain text. With private key systems since both the encryption and decryption keys are identical the key must be kept secret (or private) in order to prevent unauthorized parties from deciphering the cipher text.

Cryptographic checksum techniques use encryption in some manner to calculate the checksum. Typically a form of public key algorithms like the Rivest Shamir Adleman (RSA) scheme [RIV79] or private key algorithms like Data Encryption Standard (DES) [DEN82] in feedback mode are used to produce a 32 to 128 bit value called the checksum. The checksum can be stored with the data that was checksummed or in a safe location (safe from surreptitious modification). Using cryptographic checksums in which the checksums are stored separately is more secure in terms of forgeability. Since a cryptographic checksum requires a key, the ability to forge a cryptographic checksum is a two step process when the checksum is stored separately. First, the key must be determined, and second a different set of data (or modification of the same data), with the same checksum must be found to substitute in place of the real data. If the checksum is stored with the data, or in a modifiable location, then only the key must be known since any data with a legal checksum can replace the original data.

The use of cryptographic checksums in which the checksums are stored separately is more secure in terms of forgeability. The use of cryptographic checksums with the checksum stored with the data must be secure from known plaintext attacks and the key management must be secure. If the key is known to an attacker then it will take on the average 2^{n-1} mutations of the desired forgery to insert the forgery using the brute force attack described in section 1.3.3.1, where n is the length of the checksum in bits. That is, the checksum of each mutation has a probability of 2^{-n} of matching the stored checksum, and there is a 50% chance of a match after $\ln(2)*2^{n-1}$ mutations. In order to increase security the file can be checksummed and then encrypted to attempt to foil a plain text attack. The encryption of the file every time it is used probably would be considered undesirable on all but the fastest computers.

The drawback to cryptographic checksums is the high degree of Computational complexity of the algorithms. Encryption typically is a very computationally complex activity leading to very slow checksum computation. [HAR85] Implementing a secure cryptographic checksum using RSA can take minutes or even hours for data of a reasonable length. Cohen describes a hardware implementation with a speed of 6,500 bits/sec. [COH86] This slow speed is inadequate for practical use.

DES is less secure but much faster, especially if implemented in hardware. However, DES has the problem of private key management. Private key management is required since the same key is used to encode and decode a message. Therefore the key can not be stored where it can be accessed by an attacker. To remove access from an attacker implies that the checksum must also be inaccessible to the checksum routine. The practical implication of this is that the key must be entered each time the checksum routine is executed.

Noncryptographic checksums do not provide the same degree of security from forgery as cryptographic checksums with the checksum stored in a secure place. A noncryptographic checksum can be considered equivalent to a cryptographic checksum with a disclosed key (keys in public key encryption). Since the noncryptographic algorithm does not need to be designed to prevent discovery of the key, typically such algorithms are much less computationally complex. Being computationally less complex translates into a much faster operating speed.

1.3.3 Attacks against Checksums

In this research three types of attacks by a forger on a set of data and its generated checksum are considered. All three attacks assume that the attacker knows the checksum algorithm, can change the set of data, and can read the checksum. The three categories of attacks, which are discussed below, are the brute force attack, the birthday attack, and the trap door attack.

1.3.3.1 Brute Force Attack

A brute force attack involves generating many different sets of data until a set of data is found that has the same checksum as the original set of data. Formally, given a set of data x and a checksum algorithm f(x)=y; determine an x' such that f(x')=y. The set of data, x', which has the same checksum as the original set x, is inserted in place of the original. Because x' has the same checksum as x, it is not detected as a forgery. A more likely alternative to the generation of many sets of data is for the forger to insert the desired data into the original set of data then mutate the rest of the original data until a checksum match is found. This mutation technique allows the forger to change only small sections of the data while keeping the rest of the data unchanged. Thus the user of the data may remain unaware of the forgery because most of the data used is unchanged. If the checksum algorithm provides an even mapping, described in section 2.1.1, then a forger needs to generate on the average 2^{n-1} sets of data, where n is the number of bits in the checksum, before a checksum is found which matches the checksum of the original data. For instance, a checksum with 16 bits would require a forger to generate 32,768 sets of data before there is a 50% probability of finding a checksum match.

1.3.3.2 Birthday Attack

The birthday attack is a forgery accomplished by the originator of the data. A birthday attack involves generating many variations of an original set of data, the corresponding checksums and many variations of the set of data to be inserted and their checksums. Since any pair of original data and forged data provides a successful forgery the number of variations needed to be generated is greatly reduced. A description of the birthday attack:

1) The attacker secretly prepares a number of subtle and inconsequential changes to the valid set of data and calculates a checksum for each one.
2) An equally large number of variations of bogus data sets is generated along with the checksum for each one.
3) The checksums generated in step 1 are compared against the checksums generated in step 2.
4) If no match is found additional variations are generated until a match is found.
5) The real data set which shares the same checksum with a bogus data set is placed on the system. At a later time the bogus data set with the same checksum is substituted.

The birthday attack will succeed by producing a forgery on average after $2^{n/2}$ checksums are generated compared with 2^{n-1} for a brute force attack described in section 1.3.3.1. For a 16 bit checksum the number of checksums necessary to be generated on average for a birthday attack is only 256, compared with 32,768 for the brute force attack.

1.3.3.3 Trap Door Attacks

A trap door attack is a variation of the brute force attack. The possibility of a trap door attack occurs when the forger can invert the checksum algorithm to determine a set data that produces the same checksum as the original data. Using the checksum algorithm $f(x)=y$ a trap door exists if it is possible to determine a function $g(y) = x'$ where x' is one or more sets of data satisfying $f(x') = y$ or equivilently, $g(f(x))=x$. This $g()$ is known as the inverse of $f()$. If $g(y)$ can be determined then the checksum algorithm is susceptible to a trap door attack since the forger could generate sets of data that match the checksum of the original.

Trap door attacks are much less expensive in terms of computation effort than brute force attacks. A forgery is generated each time the inverse function is used. It is possible to not only generate forgeries, but to analyze those forgeries for their desirability as forgeries. If an attacker wishes to insert a bit pattern into a set of data at any location he would use this inverse function to generate forgeries with the same checksum as the original until the desired bit pattern occurred in one of the forgeries. Then the attacker would insert that forgery in place of the original data.

It is very difficult to show that a trap door does not exist since there is no standard method for determining if a trap door exists.

1.3.3.4 Comparison of Attacks.

Of the three attacks: brute force, birthday, and trap door, the trap door attack is the most serious. As discussed, the birthday attack is not a genuine threat in the case where the author of data is trusted. The brute force attack is good for a benchmark for general forgery, but the effort to generate a single forgery is high and the effort to generate a forgery that is useful to the attacker is very high. In contrast, the trap door attack, once a trap door is determined, is a very serious threat. The effort to generate forgeries is small compared to the brute force attack and the g(y) function can be used to generate possible forgeries until a virus is formed. Any checksum algorithm against forgery should be free from trapdoors.

1.4 Viruses

1.4.1 Description

A virus is a program that can 'infect' other programs by modifying them to include a, possibly evolved, copy of itself. [COH84] A virus typically has the following capabilities:

- identification - it can identify other files which can be modified.
- infection - it can modify zero or more of the files identified in any execution.
- action - it can take an action. The option to take an action and what action to take can be based upon the value of a trigger which is usually the satisfaction of a logical expression often based on external information, e.g. the date.

Viruses may have a "time bomb" feature such that when a logical expression is met then a specified action is taken. Such actions in recent viruses have ranged from displaying a message of world peace on the screen to reformatting the disk.

A typical virus exists as a code segment usually as the first part of a useful program. As the useful program is executed eventually the virus is executed. When the virus code segment is executed it identifies possible programs to infect (replicate itself into) then decides if it chooses to insert/append a copy of itself into the machine language code of one or more of the identified programs. When one of the newly infected programs is executed the insertion process is repeated. With the infection property, a virus can spread throughout a computer system or network using the authorizations of every user using it to infect their programs. Every program that gets infected may also act as a virus and thus the infection spreads [COH84]. The trigger mechanism of the virus is executed as part of the virus code segment. The trigger determines what additional action the virus takes. For example, on any Fridays that also fall on the 13th of the month all the files accessible to the virus are erased.

In an attempt to hide the existence and/or spread of a virus, the designers can design more complex viruses. Some of the features of more complex viruses include: insuring that files already infected are not reinfected, not infecting additional programs every time the host code of the virus is executed, mutating the code of the virus but with the desired functionality preserved, and searching for threats to the virus and disabling those threats.

Most current viruses appear to be relatively simple, but in the future more complex viruses with some or all of the features mentioned above will present threats. Though advanced viruses will present formidable threats they must draw on the resources of the computer system where they are running. Thus, viruses do not have infinite resources available to them to provide defenses or break checksum detection techniques. This lack of infinite resources makes it possible to use noncryptographic checksums to tell if a program has been infected. Otherwise, if the virus had infinite resources, the system degradation would call attention to the virus and speed its eventual eradication by system administrators.

1.4.2 Current means of control

As expected, methods of protecting data from modification also provide protection from viruses. There are several methods of protecting files against viruses. These include: access control, virus filters, snapshots, runtime models and encryption.

Access control can do much to limit the spread and damage caused by viruses. Specifically, Capability Lists, or systems with similar benefits, provide the most comprehensive protection from viruses. In a Capability List system viruses are essentially limited to only the domain in which their host program is allowed to execute. Unfortunately, capability lists exist only on a few computer systems. Access Control Lists are the dominate form of access control protection. Access Control Lists do not prevent the spread of viruses because of the large domain in which the programs operate. On a typical ACL system a program being executed by a user has the same rights as that user. Thus, a program not owned by but executed by a user can spread a virus to the user's files. Even ACL systems designed for security can allow viruses to spread [COH84].

A **virus filter** is a program that takes a suspect program and determines if the suspect program contains a virus. Deciding whether a program contains a virus is equivalent to the Halting Problem [COH84]. Therefore, writing an all encompassing virus filter is impossible.

It is possible to write a virus filter program to determine if a particular bit pattern indicative of a certain virus exists in a given program. All current virus filters work using this method. The drawback to this method is that the bit pattern of the virus must be known in advance. These simple filters will not detect any new viruses or any old viruses that have mutated.

A different type of virus filter would be able to separate programs into three classes: those programs that contained viruses, those programs that do not contain viruses, and those programs that the filter is not sure if the program does or does not contain a virus. Programs that may have a virus would then need to be examined by other methods. Such work will probably be system dependent and is at least five to ten years away.

By recording the state of the file system and examining these "**snap shots**" in conjunction with auditing records, it is possible to tell if files are being modified without permission. Such techniques can be used for virus identification after detection, but are currently not feasible for virus detection.

The **runtime** models for virus detection are Program Flow monitors and N-Version programming. [JOS88] A program can be uniquely determined by program trace information as it executes. The trace information is generated at compile time and checked against the executing program by a program flow monitor. In order for this method to work it requires a change in compiler design in order to calculate this trace information. There is also a significant runtime overhead.

N-Version Programming consists of executing several copies of a program simultaneously and followed by comparison of the outputs. This method will detect a virus if a virus has been inserted into some but not all of the copies of the program. This will not protect against fast spreading viruses where all of the copies of the program are infected. There is also a corresponding increase in overhead when compared to a single execution of the program.

Encrypting all files and only decrypting on need with a password unavailable to viruses will stop the spread of a virus. When an infected file is decrypted, the original program will be changed most likely causing a loss of functionality (especially when using cipher block chaining, see section 2.3.3.2). For frequently executed programs encryption will involve a significant increase in overhead due to the computational complexity of encryption techniques.

1.5 Problem Statement

The Clark & Wilson model appears to be the most promising approach to maintaining integrity in the commercial world. The area of the Clark & Wilson model to be used in this thesis is the prevention of change of a file as described by Clark and Wilson.

The United States Department of Defense has published a criteria for rating systems in regard to confidentiality in the Trusted Computer System Evaluation Criteria (TCSEC) [DOD85] . This document is commonly known as the Orange Book. The TCSEC provides seven levels of ratings for the ability of systems to maintain confidentiality. The ratings range from A-1, which is a verified design, through D which provides minimal protection. Though confidentiality does not automatically translate into integrity, there are many common features. Particularly, confidentiality does not protect against viruses [COH84].

The access control implemented on most systems provides weaker protection than the protection in the Orange Book rating of A or B. This gap between practice and the standard provides many opportunities for the hidden destruction of integrity. The number of commercial systems far outnumber the number of highly secure military and national systems and to date there has been less concern with the commercial system. This thesis concentrates on the commercial systems. Most of the commercial systems have some form of limited access control. An additional problem with commercial access control is that security was not considered a highly valued design criteria, thus the implementation of security tends to be less than desirable. To maintain reasonable confidence that integrity is maintained, both of these problems must be solved. In the foreseeable future, access control does not offer an adequate method to provide prevention change protection for commercial systems.

The threat to preventing change in data items is that an attacker can change the data without the user knowing it has been changed. When such a switch has occurred the user will believe the data has integrity when it actually does not. This deception occurs when the original set of data has the same checksum as the changed (or new) set of data inserted by the attacker. The attacker can either have legitimate access to change the data (but wishes to disguise the fact the data has been changed) or the attacker can be a third party who wishes to insert the forged data. The case of an attacker having legitimate access to change the data is known as a "Birthday Attack" which is described in section 1.3.3.2. This thesis is only concerned with cases where the attacker does not have legitimate access to change the data, i.e. a third party attack.

In the field of cryptography it is assumed that the attacker has unlimited current state of the art resources to employ against the encryption. This thesis does not make that assumption. Instead, it makes the assumption that reasonable resources will be expended to discover a set of data that advances the purpose of the attacker and produces the same checksum. The use of unlimited resources is unreasonable economically and, in addition, would call attention to the attack and trigger appropriate action to be taken by system authorities.

The problem this thesis will solve involves the testing of checksumming methods for use as deterrents to the integrity threat posed by viruses. General methods for constrution and testing will be developed along with developing checksum algorithms secure against viruses. The checksum algorithms to be used are variations of the QCMDCV4 [JUE86] algorithm. This algorithm and the modifications to it created for this thesis will be discussed in Chapter 2. These algorithms will be tested on DEC VAX 11/780, AT&T 3B2, and Harris HCX-9 systems and used to calculate checksums on a relatively large number of programs. The results of the checksumming will be analyzed to discover the efficiency and effectiveness of such methods. An implementation of one of these algorithms will be demonstrated using the MINIX operating system.

This remainder of this thesis is organized as follows:
Chapter 2. Error Detection with Checksums.
Chapter 3. Testing of checksum algorithms.
Chapter 4. Implementation considerations.
Chapter 5. Conclusions and further research suggestions.

Chapter 2 Error Detection with Checksums

This chapter discusses the protection provided against errors in general and against forgeries and viruses in particular of checksum algorithms. The discussion includes a general description of checksums, features of checksum algorithms including those providing protection against forgery and viruses, and methods of constructing checksum algorithms.

A checksum, or digital signature, is any fixed length block functionally dependent on every bit of the message, so that different messages will have different checksums with a high probability [DEN82]. Checksums are used to detect changes or errors in messages or sets of data between the current time and the time they were created. A checksum on a set of data is generated by a checksum algorithm. Examples of checksum algorithms include Cyclic Redundancy Codes (CRC) used in networks and cipher block chaining using the Data Encryption Standard (DES).

2.1 Features Required of General Checksum Algorithms

Good general checksum algorithms, in order to detect errors, produce checksums which have the features of even mapping, overdeterminism, and permutation sensitivity. These features are necessary in order to detect errors introduced in a set of data. [JUE86]

2.1.1 Even Mapping

Even mapping refers to the uniformity of the distribution of checksums generated by a given population of programs. The even mapping of sets of data to checksums exists if the probability of generating any given checksum is approximately equivalent to the probability of generating any other checksum over the set of all possible programs to be checksummed. One of the goals of a checksum algorithm is that given two sets of data A and B with checksums, it is desired that the checksum of A and the checksum of B be identical if and only if the sets of data A and B are themselves identical. [JUE86] Since there is many-to-one mapping from sets of data to checksums (the sets of data can be any length while the checksum is a fixed length block, and thus there are many sets of data for every checksum) the probability of two sets of data having the same checksum should not be significantly different than 2^{-n} where n is the number of bits in the checksum. A checksum algorithm which exhibits even mapping allows on the average $\ln(2)* 2^{n-1}$ sets of data that have errors or changes to occur before a set of data that is in error or has been changed is judged not to have an error or not to have been changed (probability of 2^{-n}) .

2.1.2 Overdeterminism

An overdetermined checksum algorithm is an algorithm where the resultant checksum is a function of all the bits of the set of data being checksummed. If a checksum algorithm does not provide this overdetermanism then errors that occur in bits that do not affect the checksum would not be detected by the checksum. Overdeterminism in a checksum algorithm is crucial if errors are to be detected as dictated by the even mapping feature.

2.1.3 Permutation Sensitive

A checksum algorithm is permutation sensitive if it produces different checksums for each permutation of the data elements. The permutation sensitive checksum algorithm operating on a set of data ABC produces a different checksum than the algorithm operating on permutations of that data, i.e., ACB, BAC, BCA, CAB or CBA.

2.2 Forgery

General checksum algorithms are designed to detect errors or bursts of errors that occur on a random basis. If an attacker knows the general checksum algorithm, it is relatively easy to surreptitiously insert a different set of data (a forgery) which, when using the same checksum algorithm, generates the same checksum. The two factors that increase the protection level of checksum algorithms against forgeries in general and viruses in particular are the length of checksum and the difficulty of inversion of the checksum algorithm (i.e. not having trap doors).

2.2.1 Length of the Checksum

The length of a checksum is defined as its length in bits. The checksum should be of sufficient length such that the cost of generating enough variations to find a suitable forgery (brute force attack) is unacceptably high. On the average the generation of 2^{n-1} variations is necessary to produce a set of data for forgery. [JUE86] The length of the checksum is the primary deterrent to brute force attacks.

2.2.2 Noninvertable Algorithms

A noninvertable algorithm, described in section 1.3.3.3, is a function that cannot be inverted. Thus, given a checksum and the checksum algorithm, an attacker cannot generate an algorithm that produces sets of data that, when taken as the argument of the checksum algorithm, result in the original checksum. If a checksum algorithm cannot be inverted then it has no trap doors and is not susceptible to a trap door attack.

2.3 Construction

The general techniques used to construct checksums are similar to those used in constructing ciphertext. The techniques, described below, are substitution, transposition, and feedback.

2.3.1 Substitution

Substitution involves replacing one block of data of the plaintext with a corresponding block from the ciphertext alphabet. If the message is in the plaintext alphabet $\{a_0, a_1, ..., a_{n-1}\}$ then the corresponding ciphertext alphabet is $\{f(a_0), f(a_1), ..., f(a_{n-1})\}$, where f() is a one-to-one mapping from plaintext blocks to ciphertext blocks. A simple example of substitution is to exclusive-or a constant to each character of the plaintext message to arrive at its ciphertext equivalent.

2.3.2 Transposition

Transposition is the rearranging of bits or characters according to some scheme. Transposition was classically done with aid of some type of geometric figure. [DEN82] An example given in Denning is the permutation of the characters of the plaintext with a fixed period d. A plaintext message $M=m_1 \ldots m_{d-1}\ m_d\ m_{d+1} \ldots m_{2d} \ldots$ is transposed into the ciphertext message $m_{f(1)} \ldots m_{f(d)}\ m_{d+f(1)} \ldots m_{d+f(d)} \ldots$. For example, suppose for d=4 the permutation is [DEN82]:

i:	1	2	3	4
f(i):	2	4	1	3

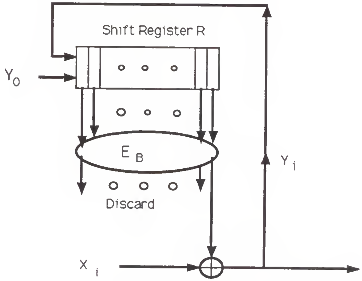
and for message M=	R E N A	I S S A	N C E
and the transposition:	E A R N	S A I S	C N E

2.3.3 Feedback

Feedback is the use of previous information in the computation of the ciphertext of the current block. This feedback mechanism can be expressed as $Y_i=f(g(X_i),Y_{i-1},Y_{i-2},...,Y_0)$ where Y_i is the ciphertext for block i, g() is the encryption function, f() is the feedback function, X_i is the plaintext for block i, and Y_0 is an initialization vector. Since the ciphertext of the last block contains information on all the previous blocks, the last block can be used as the checksum. The two most prevalent methods using feedback are Cipher Feedback Mode and Cipher Block Chaining which are discussed below along with non-linear feedback.

2.3.3.1 Cipher Feedback Mode

In Cipher Feedback (CFB) mode, ciphertext is fed back into the algorithm to generate a cryptographic bit stream, Y. A bit stream, Y_0, is used initially until there is cipher text to combine with the plaintext bitstream. Y is a function of k previous bits of the output. To obtain the ciphertext, Y, the plaintext X_i is added modulo 2 to Y_{i-1}. [JUE83] i.e. $Y_i = X_i$ ^ Y_{i-k}. The bit stream Yi may be shifted and then encrypted to enhance the security of the resultant bit stream.

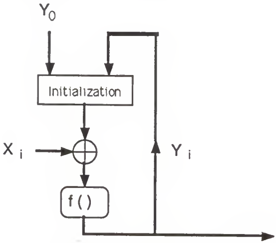


Cipher Feedback Mode (CFB)

2.3.3.2 Cipher Block Chaining

In the Cipher Block Chaining (CBC) mode of operation successive blocks of ciphertext are defined as: $Y_i=f(X_i \wedge Y_{i-1})$ where Y_0 is the initializing vector and ^ indicates bit-by-bit modulo 2 addition (exclusive-or).

Cipher block chaining is more efficient than CFB in that it uses a single execution of the block encryption algorithm for each block.



Cipher Block Chaining (CBC)

2.3.3.3 Non-linear Feedback

A nonlinear function is a function, f(), where x/f(x) is not equal to a constant. A feedback function, $f(X_i, Y_{i-1}, Y_{i-2} \ldots, Y_0)=Y_i$, which is nonlinear provides positional dependance [JUE83], i.e. permutation protection. Using non-linear feedback provides, but is not always sufficient, a method for constructing a noninvertable checksum algorithm. An example of a non-linear feedback function is: $Y_i=(X_i+Y_{i-1})^2$ modulo N, where N is a constant. The checksum is the ciphertext of the last block, Yn. Note that the addition of non-linear feedback produces a dependency of the checksum on every bit of the plaintext. [JUE83]

2.4 Algorithms

Using the methods of construction discussed in section 2.3 several checksum algorithms are presented in this section. These checksum algorithms fall into two categories: cryptographic and noncryptographic. Cryptographic algorithms require additional information, in the form of a key, to determine the checksum algorithm.

2.4.1 Cryptographic Algorithms

A cryptographic algorithm is an algorithm which requires additional information (known as a key) to be used for encryption. The role of keys will be discussed below along with examples of different types of encryption.

2.4.1.1 Keys

The function of the key is to hide the exact algorithm used from attackers. Hiding the algorithm allows the results of the cryptographic algorithm (either the ciphertext or the checksum) to be stored in a location which can be modified by an attacker. The attacker concentrates his/her efforts on determining the key to the algorithm, since, if the key is known, then any set of data and its checksum can be used as a forgery. In this scenario the legitimate user of the data generates a checksum with the cryptographic algorithm and compares it to the stored checksum (located in a modifiable location). Since the checksum of the forgery matches the stored checksum the legitimate user accepts the forgery as valid.

Cryptographic algorithms must protect the identity of the key even when an attacker knows the general cryptographic algorithm and has a copy of plaintext and its corresponding ciphertext (plaintext attack). Thus, the function $h(p,c)= k$, where p is the plaintext, c is the ciphertext, and k is the key should be computationly hard to determine. The fact that the inversion function, h(), needs to be computationly hard to determine, forces cryptographic algorithms to be computationly complex and time consuming to execute.

The alternative to storing the cryptographic generated checksum with a hidden key in a modifiable location is to use a cryptographic algorithm with a hidden key and store the checksum in a location that is not modifiable by the attacker. The attacker then would need to determine the key before attempting any of the attacks described in section 1.3.3 . This method is more secure for identifying changes in a set of data than using the power of cryptography alone.

2.4.1.2 Examples of Cryptographic Algorithms

The Rivest, Shamir, Adleman (RSA) [RIV79] cryptographic algorithm is a substitution cipher based on computing exponentials over a finite field. The RSA algorithm with cipher block chaining can be used as a checksum algorithm. The RSA algorithm is a patented public key encryption method based on the difficulty of factoring large numbers. The method has the property such that $C=M^e \bmod n$ and $M=D^d \bmod n$ with the property that ed mod phi(n) = 1, where M is the message, C is the ciphertext, e and d are the keys, n is a large prime number dependent on e and d, and phi(n) is the Euler totient function.[RIS79] The Euler totient function, phi(n), is the number of elements in the reduced set of residues modulo n. Equivalently, phi(n) is the number of positive integers less than n that are relatively prime to n. [DEN82]

RSA with large keys is very secure; key lengths of over 110 digits can be considered secure at this point in time. Using a plaintext attack the computations are on the order of exp(sqrt (ln (n) ln (ln (n)))). [DEN82] Since computational complexity is of the order of $O(n^3)$, key length is crucial to both computational intensity and security. [USE89]

The disadvantage of using RSA as a checksum algorithm is the large computational intensity of calculating a checksum. This computational complexity precludes its use for checksums on all but the fastest computers.

Cohen [COH88] suggested a method to reduce the computational complexity of using RSA for checksums. Instead of encrypting each block of data, Cohen suggested first breaking the data into larger fixed size segments. Each segment is reduced in size by using modulo division with a large prime. RSA with cipher block chaining is then applied to the reduced segments. The last block of ciphertext is used as the checksum. This method reduces the computational complexity to the computation complexity of RSA for creating ciphertext because fewer RSA block encryptions are necessary.

Cohen's original method illustrates the difficulty of detecting and preventing trap doors. In some instances, the checksum did not depend on certain parts of the file making it possible to determine a set of programs that had the same checksum [COH88]. Cohen has subsequently published a revised algorithm which corrects this problem. [COH88]

The data encryption standard (**DES**) was the official scheme approved by the National Bureau of Standards [NBS78] in 1978 to be used by federal departments and agencies for the cryptographic protection of unclassified computer data. The DES uses a block cipher method that includes a product cipher on each individual block. Formally, the DES encryption may be described as a product cipher

$$DES = (IP^{-1})J_{16} \ldots J_1(IP)$$

performed on each 64 bit block P of plaintext. IP is the bit-wise permutation with inverse IP^{-1}. The 64 bit result of the permutation is expressed as the concatenation of two 32 bit halves:

$IP(P) = L_0R_0$

$J_i(L_{i-1}\ R_{i-1})$ for $1 <= i <= 16$ is defined as:

$L_i = R_{i-1}$

$R_i = L_{i-1} \wedge f(R_{i-1},K_i)$

where K_i is derived from the secret 56 bit K, or private key.

The ciphertext is given by $C = IP_{-1}(R_{16}\ L_{16}) = DES(P)$

The source of security derives from the nonlinear many-to-one function f, which is applied to the R_i half blocks. Transposition and substitution are the main internal components of f. [MAE87] DES can be used with cipher block chaining or cipher feedback mode as a checksum algorithm.

One advantage of using the DES encryption scheme for generating checksums is that the DES algorithm is available as a chip which can be incorporated in the computer. If encryptions are generated using the DES algorithm implemented with software the process is time consuming because the DES algorithm is computationally intensive.

2.4.2 Noncryptographic Checksum Algorithms

A noncryptographic checksum algorithm is a checksum function which does not require additional information in the form of a key. Using the methods of substitution, transposition and feedback described in section 2.3, noncryptographic checksum algorithms are generated. Examples of checksum algorithms will be examined to show the specific operations that can be used in checksum algorithms. Note that some of these algorithms were not meant to be used for active forgery, or if so, appended to the end of a set of data with the resultant set encrypted. The noncryptographic checksum algorithms examined range from the simple X-OR and K-bit Linear Addition to the moderately complex Cyclical Redundancy Checksum and finally the more complex Quadratic Congruential Manipulation Detection Code (QCMDC) and its variations.

X-OR Checksum. This is a simple checksum algorithm technique which involves exclusive-oring the blocks of a message together: $Y=X_1 \wedge X_2 \wedge \ldots \wedge X_n$ where X_i is the blocks of the message. This X-OR checksum algorithm was initially proposed by the National Bureau of Standards and was in the original draft of Federal Standard 1026. [JUE83] The exclusive-or mechanism is the feedback mechanism to insure that the checksum is dependent on all the bits of the original data. This simple checksum is very susceptible to attacks such as inserting the same block of data twice while keeping the rest of the message the same ($X \wedge X \wedge Y = Y$). Additionally, blocks of data can be transposed without detection. Even if this simple checksum is added to a message which is then encrypted by DES with either Cipher Feed Back (CFB) or Cipher Block Chaining (CBC), manipulation detection is still not provided, even if the key is not known [JUE83].

K-bit Linear Addition. In this type of algorithm the blocks of data are linearly added modulo 2^k : $Y=(X_1+X_2+ \ldots +X_n) \bmod 2^k$ where Y is the resultant checksum, k is a constant, and X_i are the blocks of data. [MEY82] To forge a checksum, an attacker inserts the desired blocks while changing or reducing other blocks to match the proper checksum. The K-bit Linear Addition algorithm was designed to be used in the same manner as the X-or algorithm, i.e., a checksum generated and then the entire message encrypted. The K-bit Linear Addition algorithm provides more protection than the X-or algorithm, but it does not provide acceptable protection against manipulation of the data, especially transposition of blocks. [JUE83]

Cyclical Redundancy Checksum (CRC). This method includes a set of checksum algorithms which are widely used in detecting errors in messages passed over a network and implemented in hardware for efficiency considerations. A basic description of the process of CRC is that a polynomial of order n is chosen: $f(x)= cn*x^n+ c_{n-1}*x^{n-1} + c_{n-2}*x^{n-2} + \ldots + c_1*x^1+ c_0*x^0$ where c_i is either 0 or 1. The checksum is the block of data, n bits long, that must be concatenated to the right hand side of a set of data to be checksummed such that the combined set of data and checksum when divided, modulo two, by the chosen polynomial gives a remainder of zero. The choice of polynomial is important in detecting errors. For example, if the polynomial can be factored by (x-1) then the checksum will detect all error cases where there exists an odd number of errors. [TAN88] Typical polynomial examples include CRC-16: $f(x)=x^{16}+x^4 +1$, CRC-12: $f(x)=x^{12}+x^{11}+x^3+x^2+x^1+1$ [TAN88].

Quadratic Congruential Manipulation Detection Code (QCMDC) [JUE83] This algorithm is an example of the use of nonlinear feedback. The QCMDC algorithm is $Y_i=(X_i + Y_{i-1})^2 \bmod N$ with Y_0 an initial seed and N a large prime number. Nonlinearity is introduced by the squaring. The modular arithmetic allows the precision to be specified in advance. The QCMDC algorithm has a trap door in that it is possible to insert the desired blocks and calculate counterbalancing blocks to add in order to maintain the same checksum. For example, to insert block j between blocks i and i+1 it is necessary to determine X_k such that $Y_j=(X_j + Y_i)^2 \bmod N$ and $Y_i=Y_k=(X_k + Y_j)^2 \bmod N$. The non-linearity makes it more difficult to calculate the additional blocks to insert into the set of data than either the X-OR checksum or the K-bit Linear Addition checksum.

MDC2. This algorithm, which I created, is a variation of the QCMDC checksum algorithm. It consistes of a simple combination of exclusive-or (^), modulo division (mod), squaring (**2), addition (+), and subtraction (-) and transposition of data in a two equation format. The substitution on a byte level is provided by the exclusive-or and the addition, transposition is provided by the changing of the order of the two data terms between the two equations, nonlinearity is introduced by the squaring operation followed by the modulo division, and feedback is provided by the two equations depending on the results of the previous equations for the byte level substitution. The use of two equations reduces the ability to determine a successful trap door attack because both equations must be satisfied before a forgery can be found.

Pseudo Code for MDC2:

```
N1      = large prime a
N2      = large prime b
M1      = large prime c
M2      = large prime d
While data in file
        read first block of data into T1
        read second block of data into T2
        M1 = ((M1^T1 + M2^T2)**2) mod N1
        M2 = ((M2^T1 + M1^T2)**2) mod N2
Endwhile
```

Checksum= M1 concatenated with M2

MDC4. This algorithm, which I created, is a variation of MDC2 with increased feedback mechanisms. To reduce the possibility of construction of trap door attacks the MDC4 checksum algorithm uses four equations with four block level substitutions. The use of additional interrelated terms between the four equations increases the difficulty of finding a function that will generate an executable file from a checksum.

Pseudo Code for MDC4:

```
N1      = large prime a
N2      = large prime b
N3      = large prime c
N4      = large prime d
M1      = large prime e
M2      = large prime f
M3      = large prime g
M4      = large prime h
While data in file
        read first block of data into T1
        read second block of data into T2
        read third block of data into T3
        read fourth block of data into T4
        M1 = ((M1^T1 + M2^T2 - M3^T3 + M4^T4)**2) mod N1
        M2 = ((M2^T1 - M3^T2 + M4^T3 - M1^T4)**2) mod N2
        M3 = ((M3^T1 + M4^T2 - M1^T3 + M2^T4)**2) mod N3
        M4 = ((M4^T1 - M1^T2 + M2^T3 - M3^T4)**2) mod N4
Endwhile
```

Checksum= M1 concatenated with M2 concatenated with M3 concatenated with M4

MDC2T. This algorithm, which I created, is a variation of MCD2 with additional feedback mechanisms to defeat trap door attacks. The checksum algorithm MDC2T employs an additional substitution with feedback at the byte level. A term, tss, is formed by concatenating half of the first data block with half of the second data block. This term is used as a feedback mechanism for substitution at the block level. This additional feedback makes the task of determining trap doors more difficult.

Pseudo Code for MDC2T:

```
N1      = large prime a
N2      = large prime b
M1      = large prime c
M2      = large prime d
While data in file
        read first block of data into T1
        read second block of data into T2
        TSS= MSH of T1 ored with MSH of T2
        M1 = ((M1^T1 + M2^T2)**2+TSS) mod N1
        M2 = ((M2^T1 + M1^T2)**2-TSS) mod N2
Endwhile
```

Checksum= M1 concatenated with M2

MDC4T algorithm. This is a generalized version of the QCMDCV4 algorithm suggested by Juenman to improve upon the QCMDC algorithm. [JUE86] The true QCMDCV4 algorithm uses 32 bit blocks resulting in a 128 bit checksum and set values of the primes and initial seeds. The MDC4T checksum algorithm has the general form of the QCMDCV4 algorithm but can be used with shorter block lengths to facilitate efficient computation. In order to introduce additional non-linearity, substitution was added to the QCMDC algorithm which only uses feedback. The substitution was provided by exclusive oring intermediate checksum totals to the data before use. To prevent trap doors a transposed history function was added. The result is that there are multiple different references to previous blocks that would need to be satisfied in order to surreptitiously insert blocks of data. The MDC4T algorithm is:

```
N1      = large prime a
N2      = large prime b
N3      = large prime c
N4      = large prime d
M1      = large prime e
M2      = large prime f
M3      = large prime g
M4      = large prime h
While data in file
        read first block of data into T1
        read second block of data into T2
        read third block of data into T3
        read fourth block of data into T4
        TSS= MSQ of T1 ored with MSQ of T2 ored with MSQ of
                T3 ored with MSQ of T4
        M1 = ((M1^T1 + M2^T2 - M3^T3 + M4^T4)**2+TSS) mod N1
        M2 = ((M2^T1 - M3^T2 + M4^T3 - M1^T4)**2-TSS) mod N2
        M3 = ((M3^T1 + M4^T2 - M1^T3 + M2^T4)**2+TSS) mod N3
        M4 = ((M4^T1 - M1^T2 + M2^T3 - M3^T4)**2-TSS) mod N4
Endwhile
```

Checksum= M1 concatenated with M2 concatenated with M3 concatenated with M4

The QCMDCV4 algorithm appears very strong in terms of defeating forgery attacks in that it provides noninvertability and at 128 bits is long enough to defeat birthday attacks. [JUE86] The MDC4T algorithm maintains the noninvertability aspect, but for checksum lengths of less than 128 bits, does not protect against birthday attacks. [JUE86]

2.4.3 Comparison of Cryptographic and noncryptographic algorithms

The theoretical difference between cryptographic and noncryptographic algorithms is that with cryptographic algorithms the attacker does not possess the total algorithm and thus cannot perform the attacks described in section 1.3.3. A drawback to cryptographic algorithm is that the key must be provided to the checksum algorithm each time the algorithm is to be used.

A practical disadvantage of cryptographic algorithms is that they are designed to conceal the identity of the key. This makes cryptographic checksums very complex computationally, effectively eliminating their use on present microcomputers.

The strengths of cryptographic algorithms are in the substitution and transposition of blocks of data. Typically little is provided in terms of feedback mechanisms. Noncryptographic algorithms generally provide little (compared to cryptographic algorithms) in terms of substitution or transposition, but provide very strong feedback mechanisms. For example, DES encryption alone without feedback requires, per 64 bit block of data, 2 transpositions each of 64 bits, 16 transpositions each of 32 bits, 16 transpositions combined with substitutions each of 48 bits, 16 transpositions with substitution each of 48 bits, 16 permutations each of 32 bits, and 16 permutations each of 48 bits. In contrast, MDC4, using 16 bit blocks with a 64 bit checksum, has 32 substitutions each of 16 bits, and 16 transpositions each of 16 bits.

Since nonlinear feedback is the primary mechanism to prevent trapdoors and because of the large computational complexity of cryptographic algorithms, this research has focused on noncryptographic algorithms. The noncryptographic algorithms MDC2, MDC2T, MDC4, and MDC4T were selected for further study because of their ability to provide protection from forgery while providing efficient execution on small computers.

2.5 Conclusions

In this chapter we have described the features that a checksum algorithm must have in order to detect errors and to defeat attempted forgeries by an attacker. These features include even mapping, permutation sensitivity, overdeterminism, length and noninvertablity. A general basis for construction of these checksum algorithms was provided and examples of both cryptographic and noncryptographic algorithms presented. Noncryptographic checksum algorithms were shown to be better for detection of change in the small computer environment because of their lower computationally intensity. Four noncryptographic algorithms (MDC2, MDC2T, MDC4, MDC4T) were chosen for further study and testing in chapter 3.

Chapter 3 Testing of Checksum Algorithms

This chapter describes methods of testing checksum algorithms. These testing methods are broken into three areas: statistical tests for even mapping, mutation tests for forgery protection, and computational complexity tests for efficiency. The checksums tested were the MDC2, MDC2T, MDC4, MDC4T with resulting 32 bit checksums.

3.1 Statistical Tests

Statistical tests are used to determine if a checksum algorithm produces checksums that map evenly over the range of the checksum, i.e. the checksums are evenly distributed between the range 0 and 2^n-1, where n is the number of bits in the checksum. The even mapping of checksums produced by a checksum algorithm is important because of the protection it provides against brute force attacks. The method we use to accomplish this test is to use the null hypothesis that the distribution of checksums from a checksum algorithm is an even distribution with the alternate hypothesis that the checksum distribution is not evenly distributed. The chapter is organized into the following sections: descriptions of the statistical tests, description of the generation of simulated programs, and the results of the statistical tests.

3.1.1 Description of Statistical Tests

This null hypothesis is tested using several statistical tests including Chi-square, Collision, and Binomial tests.

3.1.1.1 Chi-square Test.

The chi-square test which is based on the chi-square statistic provides a measure of the goodness of fit between observed data and the expected values of that data. The chi-square statistic is used to attempt to show that the null hypothesis, that the checksums produced by a checksum algorithm are randomly distributed, is contradicted by the data. The chi-square statistic is also used to determine the statistical significance of results of other statistical tests. In this instance, the chi-square statisticis employed to evaluate the results from the binomial test.

Chi-square (X^2) statistic is a measure of the difference between the observed value and the expected value. The chi-square statistic is expressed as:

$$U = \sum \text{(observed - expected)}^2 / \text{expected}$$

The statistic, U, of a chi-square test is examined to determine the confidence we have in the fit that U describes. In order to evaluate the value of U, the chi-square statistic it is necessary to know the number of degrees of freedom. For our applications, the number of degrees of freedom is one less than the number of possible outcomes.

For a large number of degrees of freedom the following values are calculated using the formula given in Knuth [KNU81]:

$$X^2 = v+(2v)^{.5} x_p + (2/3)*(xp^2 -1)+ O(1/v^{.5}) \qquad \{1\}$$

where xp=1%:-2.33, 5%:-1.64, 25%:-.675, 50%: 0, 75%: .675, 95%: 1.64, 99%: 2.33

If one has 99 degrees of freedom the results of calculating a value for {1} is:

	p=.01	p=.05	p=.25	p=.50	p=.75	p=.95	p=.99
v=99	69.23	77.04	89.14	98.33	108.14	123.23	134.64

where p is the probability that the result, or a more extreme (unlikely) result could have occurred under the null hypothesis. If p is small then either an extreme (unlikely) event has been measured or the null hypothesis is false.

It is desirable that there be five or more expected observations per category [KNU81], therefore checksums are sorted into a smaller number of distinct categories. For this work a value of 100 categories was chosen, resulting in 99 degrees of freedom.

The chi-square statistic is excellent in examining the overall distribution of the checksums.

3.1.1.2 Collision Test

The Collision test is applicable when the number of possible outcomes of observations is much larger than the number of observations taken. For instance, suppose there are m urns and we throw n balls at random into those urns, where m is much greater than n. Most of the balls will land in urns that were previously empty, but if a ball falls into an urn that already contains at least one ball we say that a "collision" has occurred [KNU81].

If the checksums generated by the checksum algorithm map evenly over its domain (the null hypothesis) then it should be possible to predict the number of collisions (multiple observations of a checksum). The number of collisions is dependent on the number of observations taken (programs checksummed), m, and a number of possible values those checksum can take, n (for a 32 bit checksum n is 2^{32}).

The probability that a given possible checksum will contain exactly k observations is:

$$pk=\binom{n}{k} m^{-k} (1-m^{-1})^{(n-k)}$$

so the expected number of collisions (multiple observations of a possible checksum) is:

$$\sum_{k>=1} (k-1)p_k = \sum_{k>=0} (k*p_k - \sum_{k>=1} (p_k)) = n/m-1+p_0. \text{ since } p_0=(1-m^{-1})^n = 1-n/m + \binom{n}{} m^{-2}$$

+smaller terms

Evaluating the equation shows that the average total number of collisions taken over all m checksums is very slightly less than $(n^2)/2m$ [KNU81]. For a 32 bit checksum and 512,000 observations the expected number of checksum collisions is 30.5.

A table of expected probabilities of c collisions occurring is the probability that (n-c) checksums are generated with m tests and n possible checksums, i.e. $(m*(m-1)* \ldots * (m-n+c+1))/(m**n) * \left\{ {n \atop n-c} \right\}$.

An approximation [KNU81] of the probabilities for different numbers of collisions, c, are shown below.

Probability	.99	.94	.71	.44	.24	.05	.01
Expected Collisions	43	39	33	29	26	21	1.7

3.1.1.4 Binomial Test

In the null hypothesis that checksums map evenly over the interval, the number of one bits in checksums should follow a binomial distribution. The observed number of one bits can be compared to the expected number based on an even mapping of checksums and the difference can be tested for significance using the chi-square statistic.

If there is an even mapping, the probability of any bit in a checksum having a value of one is .50. Thus the expected probability distribution is given by the formula:

$$p(x) = \binom{32}{x} (.5)^{x} (1-.5)^{(32-x)}$$

where x is the number of one bits in the checksum. The observed versus the expected results are evaluated for significance using the chi-square statistic with the appropriate degrees of freedom.

Since the expected value for observations at the ends of the scale is close to zero the number of degrees of freedom for the chi-square test is reduced. Chi-square values for 26 and 36 degrees of freedom:

	.01	.05	.25	.50	.75	.95	.99
v=26	12.15	15.30	21.50	26.00	30.50	38.95	45.75
v=36	19.18	23.21	29.91	36.00	41.36	51.04	58.72

3.1.2 Simulation of Executable Programs

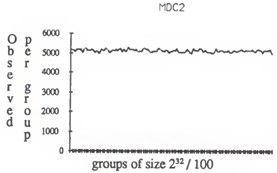
In order to provide a sufficient number of programs to be able to test the properties of the checksum algorithms it was necessary to simulate a series of executable programs. These simulated executable programs were generated by concatenating a series of random numbers. The end of the program was determined by a specific terminator string chosen at random thus providing programs of varying lengths. Since the addition of executable statements at the end of the program constitute a new program, computation time was reduced by generating new programs.

3.1.3 Results of Testing

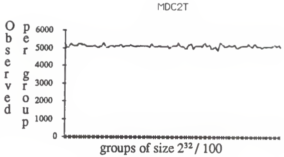
This section presents the results of the statistical tests, Chi-square, Kolmogorv-Smirnov, Collision and Binomial tests, when applied to the four checksum algorithms MDC2, MDC2T, MDC4, MDC4T.

3.1.3.1 Chi-square Test.

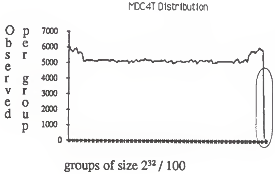
The chi-square test tests the whether to checksums generated by a checksum algorithm are evenly distributed. The checksums for 512,000 observations partitioned into 100 distinct equal sized categories are graphically displayed below:



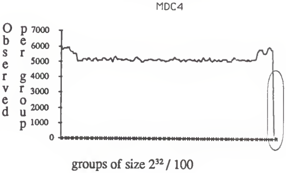
MDC2



MDC2T



MDC4T Distribution



MDC4

Observed number of checksums per group, assuming even distribution, is 5120

After segmenting the checksums into 100 even groups the chi-square and their corresponding p - values for each of the algorithms calculated:

	chi-square	p - value
MDC2:	106.27	.70
MDC2T:	101.97	.59
MDC4:	11347	1.00
MDC4T:	11419	1.00

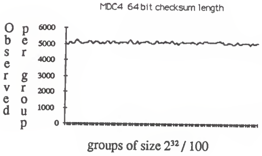
The chi-square values for the MDC2 and MDC2T are not in an acceptable range for rejecting the null hypothesis that the checksums are evenly distributed. The chi-square values for MDC4 and MDC4T clearly provide evidence against the null hypothesis.

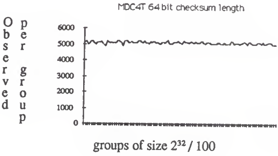
This is also indicated by the graphical representation, especially at the high end points of the interval and is the circled areas on the MDC4 and MDC4T graphs.

The reason MDC4 and MDC4T have high chi-square values is because of the prime numbers used in the algorithm. Because the testing was done for 32 bit checksums, the prime numbers used for modulo operation were significantly less than 2^8. Furthermore, in order to reduce the probability of trap door attacks, different primes were chosen, even further eliminating potential checksum. In the graph of distribution of checksums, it is clear that there is a significant decrease of observed checksums at the maximum possible checksum.

Further tests were conducted using 64 bit checksums to determine if the choice of prime numbers used in the 32 bit MDC4 and MDC4T algorithms were the reason for the large chi-square values or if there is an inherent flaw in those algorithms. The results for 64 bit MDC4 and MDC4T checksums were determined:

	chi-square value	p - value
MDC4:	89.1	.25
MDC4T:	119.6	.90





The values obtained for the MDC4 and MDC4T 64 bit checksum algorithms do not reject the null hypothesis.

3.1.3.2 Collision Test

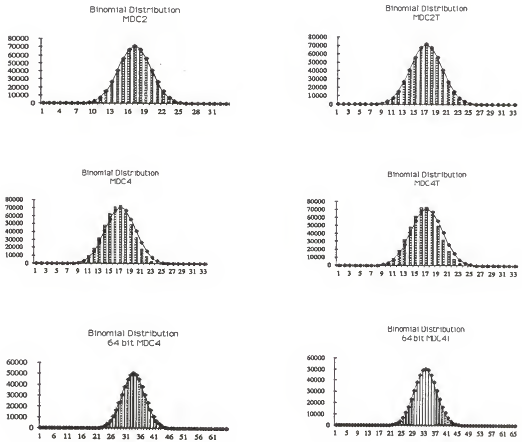
The collision test measures how many times there are multiple occurrences of a checksum (collisions) in a series of observations. With 512,000 observations the following results were obtained:

	Collisions Observed	P - value
MDC2:	25	.20
MDC2T:	33	.71
MDC4:	33	.71
MDC4T:	38	.90

The collision test results for all four checksum algorithms indicate no evidence to reject the null hypothesis.

3.1.3.4 Binomial Test

The binomial test tests to see if each bit of a checksum had a .50 probability of being a one. The graphical results of observed versus predicted are shown below.



X axis: # of one bits; Y axis: #of checksums, out of 512,000, which contain that number of one bits
The bars show the actual values; the lines are the expected values assuming even distribution

Chi-square values for 26 and 36 degrees of freedom:

	.01	.05	.25	.50	.75	.95	.99
v=26	12.15	15.30	21.50	26.00	30.50	38.95	45.75
v=36	19.18	23.21	29.91	36.00	41.36	51.04	58.72

	chi-square value	p - value
MDC2:	31.196	.77

MDC2T:	26.071	.50
MDC4:	19,267.	1.00
MDC4T:	19,346.	1.00

The chi-square statistics from the binomial test for the MDC2 and MDC2T algorithms do not provide evidence to reject the null hypothesis. The chi-square values for the MDC4 and MDC4T algorithms provide evidence to reject the null hypothesis.

Along with the 32 bit MDC4 and MDC4T the 64 bit MDC4 and MDC4T were tested using the binomial test. The following results were obtained:

	chi-square value	p - value
64 bit MDC4:	47.2	.87
64 bit MDC4T	39.0	.61

The chi-square statistics for the binomial tests on 64 bit MDC4 and MDC4T do not provide evidence to reject the null hypothesis.

3.1.4 Conclusions from statistical testing

The results of the chi-square, collision, and binomial tests are used to test the null hypothesis, that the checksums are evenly distributed.

For the MDC4 and MDC4T checksums algorithms with 32 bit checksums there is evidence to reject the null hypothesis that the checksums are evenly distributed. Of the three tests only the collision test provided evidence of even mapping. The results of the other two tests (chi-square and binomial) provide evidence to reject the null hypothesis. Extending the MDC4 and MDC4T algorithms to 64 bit checksums provides statistical results which do not provide evidence to reject the null hypothesis. Thus we conclude that it is the small relative primes used in the 32 bit four equation checksums that causes the uneven distribution and not the form of the algorithm.

The null hypothesis cannot be rejected with the MDC2 and MDC2T checksum algorithms. The three tests; chi-square, collision and binomial do not provide evidence to reject the null hypothesis.

3.2 Mutation Testing

To simulate an actual attack on a given checksum a mutation test was used. Given a file of blocks: $F= F_0,F_1,...,F_m,F_{m+1},...$ the attacker attempts to determine a V consisting of $V_0,V_1,...V_n$ such that when it is inserted into the file with the resulting file, $F' = F_0,F_1,...,F_m,V_0,V_1,...,V_n,F_{m+1},...$ the checksum of F' is equal to the checksum of F. This insertion attack can be further specified that given a checksum function C() the attacker must find a V such that

$$C(F_0,F_1,...,F_m) = C(F_0,F_1,...,F_m,V_0,V_1,...V_n)$$

Note: If the checksum algorithm is invertable (in the manner that if given a resultant checksum Y_i , block of data X_i, and the checksum algorithm C() the value of Y_{i-1} can be found) this is equivalent to the birthday attack.

In a real attack the first part of V would be the virus while the last part would be filler to make the equation above true. A mutation attack determines how many different fillers must be examined before finding a filler that makes the equation above true.

In a test of approximately 64,000,000 (2^{24}) mutations, no mutation was found such that C(F) was equal to C(F') for any of the checksum algorithms MDC2, MDC2T, MDC4, MDC4T. The CPU time necessary to check 224 mutations was 624 seconds on a Harris HC-9 computer. Extrapolating this result to the time necessary (in CPU seconds) to generate a forgery with following percentage probability:

12.5%	25%	50%	75%	87.5%
21,300	46,000	110,700	221,500	332,200

For virus protection on small computers these times should provide adequate protection against a mutation attack.

3.3 Efficiency Testing

The efficiency test used in this work is the time a given checksum algorithm will generate a checksum for a set length file. In order for a checksum algorithm to be used it must execute in a reasonable amount of time. The time taken for file access and checksum algorithm was tested on three different types of computers within the IBM PC family. The computers tested were:

1) IBM PC with 8088 processor, 4.77 MHz clock speed, 20 megabyte hard drive with an access time 80 milliseconds. This type of computer represents the slowest type of machine on which checksum protection can be expected to be used.

2) IBM PC clone with 8088 processor, 10 MHz clock speed, 40 megabyte hard drive with an access time of 65 milliseconds. This computer represents the current entry computer.

3) IBM PC clone with 80286 processor, 10 MHz clock speed, 40 megbyte hard drive with 28 millisecond access time. This type of computer represents the current mid-to-high range. In the future this type of computer will represent the entry computer.

The algorithms tested were the 32 bit versions of MDC2, MDC2T, MDC4, MDC4T. The implementations of these algorithms were written in the computer language C, and compiled with the Turbo C compiler.

Results, in seconds, for 32 bit checksum:

Computer:	1	2	3
MDC2	32.9	17.1	4.6
MDC2T	40.4	21.1	4.7
MDC4	62.8	32.3	7.9
MDC4T	63.5	32.5	7.9

Results, in seconds, for 64 bit checksum:

MDC4	40.2	19.0	4.7
MDC4T	40.2	21.1	4.9

The dramatic differences between the times for these three machines are mainly due to three factors: disk access speed, clock speed, and different execution times for operations.

Disk Access: In order to identify the differences between algorithms a copy of the tested programs without a null code checksum algorithm was run on the different computers. The results were:

	time	published access speeds
computer 1:	10 seconds	80ms
computer 2:	7.4 seconds	65ms
computer 3:	2.4 seconds	28ms

This difference can be explained by the speed of disk access and represents a lower bound to the speed of which any algorithm that examines the entire executable file can execute.

Clock Speed: The speed of the CPU is an important factor in the elapsed time to execute a checksum algorithm on a program. The faster cycle time is the primary difference between the two 8088 based machines.

Execution Time for Operations: Certain instructions take significantly less time to execute on a 80286 (and 80386) processor than on the 8088 processor. The two instructions where the speed was increased significantly were division and multiplication. The number of clock cycles it takes for the 8088 processor to execute a divide is between 144 and 166 (for 16 bit divides) compared with 22 for a 80286 or 80386 processor. Since divide operations are a crucial operation for the modulo operation and are done 6 times for each checksum in the MDC2 and MDC2T algorithms and 12 times in the MDC4 and

MDC4T algorithms, division greatly increases the time necessary for execution. A similar difference can be observed for multiplication which occurs two or four times in calculating a checksum depending on the algorithm.

3.4 Conclusions

The checksum algorithms were examined from the perspectives of providing an even mapping of checksum, insertion of filler data such that the same checksum is obtained, and the speed of execution. The two algorithms, MDC4 and MDC4T with 32 bit checksums do not provide even mapping, and thus can be eliminated. In their place we also considered the MDC4 and MDC4T algorithms with 64 bit checksums. These two algorithms with 64 bit checksums did provide even mapping and are acceptable checksum algorithms.

All the algorithms, as expected from an even mapping perspective, provided protection against a mutation attack.

The efficiency of the checksum algorithms varied greatly with respect to computer. The rank ordering of efficiency was: 1) MDC2, 2) MDC2T, 3) MDC4 (64 bit) and 4) MDC4T (64 bit). On slower machines (and especially those with 8088 processors) the differences are significant. On 80286 (and further generations of the 80x86 family) processor based machines the differences in execution speed are not significant.

Chapter 4 Implementation

A virus detection program should reliably inform the user that a virus has entered the system. This function can be broken into two parts: detecting the virus and informing the user. Both of these functions rely on the operating system to insure the virus detection code is executed before each program is started and that the virus detection code and checksum value has not been changed. Unfortunately, with most microcomputers the operating system provides only minimal protection at best.

4.1 Goals

The goals for implementation of a virus detection mechanism using checksums are: 1) that the virus detection mechanism is executed before each program is executed and that the virus detection mechanism is not changed by a virus, 2) that the checksum generated by running the checksum algorithm is stored in a place not easily modifiable to the user, and 3) that the virus detection feature can be implemented without major changes in system operation.

4.1.1 Protected Operating System/Checksum Routine

The change of an operating system to execute a virus detection program before the execution of user programs is relatively simple. Insuring that the code that calls the virus detection program and that the virus detection program has not been changed is difficult on small and personal computer operating systems. If a virus knows the location and operation of the virus detection code and has the ability to change that code, it is possible for a virus to disable the virus detection routine. An example of change which would nullify the work of a detection program would be to modify the return value of the comparison of old and new checksums such that the return value always indicated no virus.

4.1.2 Protected Checksum Storage

The checksums for programs should be stored in a location not readily accessible to the user or to the virus operating on behalf of a user. If a virus has "write" access to the location used to store checksums and knows the checksum algorithm, the virus can calculate a checksum on the virus infected program and insert the new checksum so that the virus detection algorithm does not detect the virus.

4.2 Protection Features

Ideally users would be prevented from modifying either the virus detection code or the checksums. This implies that the user's programs are limited to accessing only the memory allocated to them. Methods of limiting programs to set memory ranges include bounds checking, and virtual memory.

4.2.1 Bounds Checking

A program that employes bounds checking compares each requested address with the bounds register(s). If the address is not within the acceptable accessible memory the operation is not allowed to be executed and the job terminated with the proper error message.[DIE84] Bounds checking can be done on all levels of memory and most particularly main memory and secondary (disk) storage. Most microcomputers do not employ bounds checking, thereby severely limiting its use in protection against modification of the virus detection routine by viruses.

4.2.2 Virtual Memory

The typical microcomputer is designed to be a single user system with that user having total control over all available memory locations. These systems can use a virtual memory operating system.

When using an operating system employing virtual memory, each user process has a private address space that contains its programs and data. Each word in the process's address space has a fixed virtual address that the programs in the process use to access that word. In executing a memory reference instruction, the hardware computes the virtual address that identifies the target location of the reference by using a value or offset contained in a field of the instruction plus some index registers and address registers. The virtual address is then translated, or mapped, by hardware into a physical address. This translation is transparent to the program. [GEL88] The translation from virtual addresses to physical addresses can be as simple as adding a nonmodifiable base value to all virtual addresses giving physical addresses to demand paging or segmentation schemes.

Virtual memory systems are slower than nonvirtual memory systems because there is at least one address conversion for each memory reference. Even if these functions are implemented in hardware or microcode there is a performance penalty. For this and other reasons most microcomputers do not have virtual memory implemented by their operating systems.

4.2.3 Ignorance

One of the strongest protection features is the lack on knowledge on the part of the virus of the exact location of the virus detection code. If the virus does not know where the virus detection code is stored, then it must either search for that code or modify code at random.

For the virus to search for the virus detection portion of the kernel, it must have a pattern to search against. This implies that the virus designer knew a good deal about the virus detection code and that the virus will carry around enough tell tale parts of the virus detection code to be able to identify the virus. Since the virus detection code is not trivial this often increases the size of the virus significantly. If this threat is considered serious enough then multiple copies of the virus detection code can be used. A further step is to have different implementations of the virus detection code in several different locations so that a virus would need to carry information on each virus detection code in order to disable all of the virus detectors.

4.3 MINIX Example

MINIX is an operating system that is a subset of UNIX Version 7 (V7). MINIX was developed by Tanenbaum and is described in his Operating Systems textbook. [TAN87] MINIX contains nearly all the V7 system calls, and these calls are identical to the corresponding V7 calls. MINIX was originally written for the IBM PC, XT, and AT and has since been ported to the NS 16032 and the 68000. The version of MINIX used in this research work is version 1.1 for the IBM XT. For further details refer to the textbook which includes most of the operating system source code.

4.3.1 General Description

MINIX is a layered operating system where communications between layers is accomplished by message passing thus insulating the kernel from the users.

When a process is created its cs register is set at the base address of the process. The process is allocated the amount of space specified in a header file at the top of the program to be run. This amount of space is typically 64K.

There is no checking for attempts to read or write outside the memory requested. All addresses are physical addresses (no virtual memory) and instructions can read or write areas in the operating system space by changing the register values.

4.3.3.1 MINIX Protection Mechanisms

MINIX protection mirrors UNIX protection and is a variation of an Access Control List based system. Each user has a domain that it can operate in (files it has certain access rights to) defined by its userid (uid) and group id (gid). If an object is not in the domain of a process then the process is refused access to that object. The rights that processes may possess are read, write, and execute.

4.3.2 Implementation

For the purposes of this thesis, it was deemed that the Operating System provides adequate protection for both the checksum routine and the storage of checksums. The checksum will be stored in a file readable by all, appendable by users, and changable by root.. When a process requests to execute a new procedure a execve call to the operating system is made. The calling process passes the name of the procedure along with the proper checksum is passed. The do_exec procedure in the memory manager calculates a checksum and compares it with the checksum passed. Different values for the calculated and the passed checksums terminate execution of the program.

4.3.4 Weaknesses

The weakness of this approach is not being able to limit user processes to proper memory locations. Since there is no bounds checking or virtual memory, a virus potentially has ability to change any memory location.

4.4 Suggestions for other Operating Systems

The weaknesses of MINIX are present in all operating systems that do not isolate a user process in its own memory area. Most operating systems do not provide even the protection mechanisms of MINIX. Thus any file can be modified or executed by the user or a virus acting on his/her behalf, instead of just those the process has "write" access.

Ideally, users will have the operating system source code to directly incorporate the virus detection mechanism, and then recompile the operating system. This is not the case with most operating systems. Instead the virus detection mechanism must be added on top of the operating system. Placing the virus detection mechanism outside the operating system makes its location better known and easier to disable.

4.5 Conclusions

Modest protection by checksums can be provided even with an insecure operating system. However, with these operating systems a virus can either attack the checksum mechanism or determined programs with the same checksum. When a virus is limited to the section of memory it is allocated (with either bounds checking or virtual memory) then only brute force or trap door attacks are feasible.

Chapter 5 Conclusions

This chapter provides a brief review and conclusions from the previous four chapters, draws conclusions with respect to particular classes of computers, and discusses future research possibilities.

5.1 Review

The virus problem is considered a subset of a larger integrity issue. Virus detection/prevention can be directly classified under the control of change of static data in the Clark & Wilson integrity model. The only method that currently shows promise for detecting any virus other than the simplest virus is a checksum technique. These checksums can be generated by either cryptographic or noncryptographic algorithms.

Checksum algorithms must have the properties of even mapping, permutation sensitivity, and overdeterminism. To provide protection against an active attacker versus detecting random errors, a checksum algorithm must produce checksums that are of adequate length and the algorithm must be noninvertable. The active attacker can employ several different types of attacks including the brute force attack and the trap door attack. Another attack, the birthday attack was deemed not applicable to the virus problem when a strong checksum algorithm is employed The trap door attack was deemed to be the most serious threat.

Checksum algorithms employ the techniques of substitution, transposition and feedback to produce checksums that provide the necessary strength to deter attackers. Both cryptographic and noncryptographic checksum algorithms employ these mechanisms. The cryptographic algorithms typically employ large amounts of substitution and transposition making the algorithms very computationally complex. The computational complexity of cryptographic algorithms limits their use to fast computers. Noncryptographic algorithms can provide adequate protection against attackers with fast enough execution to use on small computers.

Four specific noncryptographic algorithms were investigated. The tests employed included statistical, efficiency and a simulated attack. Two algorithms, MDC2 and MDC2T were shown to provide adequate protection with 32 bit or greater checksum length, while two other algorithms, MDC4 and MDC4T, provided adequate protection at the 64 bit checksum length.

5.2 Particular Conclusions

This section discusses the effects of the general conclusions as they apply to specific classes of computers.

The basic trade off with noncryptographic algorithms is efficiency versus trap door protection. The trap door protection is provided by additional substitution and feedback as described in section 2.3. The differences in feedback between the four algorithms discussed were either by adding an extra history term (tss) or by increasing the number of equations for which a single data block is directly used (four equations versus two equations).

The additional feedback provided by the extra tss term in the MDC2T and MDC4T algorithms should be more resistant to trapdoor attacks than the corresponding algorithms without the tss term (MDC2 and MDC4). For example, the MDC2T algorithm should be more resistant to trap door attacks than the MDC2 algorithm.

The four equation algorithms, MDC4 and MDC4T, should provide more protection from trapdoor attacks than two equation checksum algorithms, MDC2 and MDC2T. When using 16 bit data blocks with the MDC4 and MDC4T algorithms (64 bit checksums) there should not be a decrease in the effort to determine trap door attacks.

For computers that have fast disk access and low CPU cycle time the use of the MDC4T (64 bit) algorithm is suggested. The additional protection against trap door attacks is provided with only a small time penalty. For computers that have medium disk access time and medium CPU cycle time the recommended choices are the MDC4T (64 bits) for best protection or the MDC2 for faster execution with less protection. For slow computers the MDC2 algorithm is recommended.

These results are summarized in the table below.

		Disk Access Time	
		Fast	Slow
Computer Speed	Fast	MDC4T	MDC4T
	Slow	MDC2	MDC2

As a review, the basic forms of the algorithms:

MDC2: Two equations of the form
M1 = (M1^T1 + M2^T2)**2 Mod N

MDC2T: Two equations with additional feed back term of the form
M1 = (M1^T1 + M2^T2-TSS)**2 Mod N

MDC4: Four equations of the form
M1 = (M1^T1 - M2^T2 + M3^T3 - M4^T4)**2 Mod N

MDC2T: Four equations with additional feed back term of the form
M1 = (M1^T1 - M2^T2 + M3^T3 - M4^T4-TSS)**2 Mod N

5.3 Further Research

The virus detection/protection field offers areas of future research. It is desirable to be able to prevent viruses from entering a computer system by examining the entering information. Though virus detection is undecidable in the general case, it may be possible to partition programs into one of three categories: 1) program does not contain a virus, 2) program contains a virus, and 3) cannot tell if the program does or does not contain a virus. If the third category can be reduced to a modest level this would represent significant progress in virus protection. Note that this would probably need be done at the object code level.

The integrity field is a fertile area for future research. There is a need for work at all levels including:

1) General Models. The models for integrity are generally considered inadequate at the same time the need for integrity is increasing. Since lower level models depend on theoretically sound higher level models advances in this area are important.

2) Intermediate Concerns. The identification of integrity mechanisms that are common across most or at least many applications are needed. These mechanisms provide the

building blocks to enable applications to maintain integrity.

3) Implementation Concerns. The actual implementation and study of the use of general integrity mechanisms is needed.

References

[BEL75] Bell, D. E., La Padula, L. J., Secure Computer Systems: Unified Exposition and Multics Interpretation, MTR-2997, The Mitre Corporation, 1975.

[BIB77] Biba, K. J., Integrity Considerations for Secure Computer Systems, USAF Electronic Systems Division, ESD-TR-76-372, 1977.

[BOE85] Boebert, W. E., Kain, R. Y., A Practical Alternative to Hierarchial Integrity Policies, 8th National Computer Security Conference 1985, 18-27.

[CLA87] Clark, D. D., Wilson, D. R., A Comparision of Commercial and Military Computer Security Policies, IEEE Security and Privacy 1987, Oakland, CA, 184-194.

[CLA89] Clark, D. D., Wilson, D. R., Evolution of a Model for Computer Integrity, Invitational Workshop on Data Integrity, 1989.

[COH84] Cohen, F., Computer Viruses, PhD Dissertation, 1984.

[COH86] Cohen, F., A Cryptographic Checksum for Integrity Protection, Computers and Security 6 1987, 505-510.

[COH88] Cohen, F., On the Implications of Computer Viruses and Methods of Defense, Computers & Security 7 (1988),167-184.

[DAV84a] Davies, Donald W., A Message Authenticator Algorithm Suitable for a Mainframe Computer. CRYPTO 84 (1984), Santa Barbara, CA, 393-400.

[DAV84b] Davies, D. W. , Price, W. L., Security for Computer Networks, John Wiley & Sons, New York, 1984.

[DEI84] Deitel, H. M., Operating Systems, Addison-Wesley Publishing Company, Reading, MA, 1984.

[DEN82] Denning, D., Cryptography and Data Security, Addison-Wesley Publishing Company, Reading, MA, 1982.

[DES85] Desmedt, Y., Unconditionally Secure Authentication Schemes and Practical and Theoretical Consequences, CRYPTO 85 (1985), 42-55.

[DES87] Desmedt, Y., Is There an Ultimate Use of Cryptography, CRYPTO 87 (1987), Santa Barbara, CA, 459-463.

[DOD85] Department of Defense National Computer Security Center, Department of Defense Trusted Computer System Evaluation Criteria, DoD 5200.28-STD, 1985.

[DIF76] Diffie, W. and Hellmen, M, New Directions in Cryptography, IEEE Transactions on Information Theory 1976, 644-654.

[GAS88] Gasser, M., Building a Secure Computer System, Van Nostrand Reinhold, New York, 1988.

[GIF82] Gifford, D. K., Cryptographic Sealing for Information Secrecy and Authentication, Communications of the ACM, April 1982, V 25 #4, 274-286

[HAR85] Harari, S., Non Linear Non Commutative Functions for Data Integrity, CRYPTO 85 (1985), Santa Barbara, CA, 25-32.

[HEN87] Henning, R. R., Walker, S. A., Data Integrity vs Data Security: A Workable Compromise, NCSC National Computer Security Conference 1987, 334-339.

[HSI 79] Hsiao, D. K., Kerr, D. S., Madnick, S. E., Computer Security, Academic Press, New York, 1979.

[HUA88] Huang, Yue Jiang and Cohen, Fred, Some Weak Points of One Fast Cryptographic Checksum Algorithm and its Improvement. Computers & Security, 7 (1988), 503-505.

[JOS88] Joseph, M. K., Avizienis, A., A Fault Tolerance Approach to Computer Viruses, IEEE Conference on Security and Privacy 1988, p 52-58.

[JUE83] Jueneman, R. R., Matyas, S. M., Meyer, C. H., Message Authentication with Manipulation Detection Codes, IEEE Conference on Security and Privacy 1983, 33-54.

[JUE86] Jueneman, R. R., A High Speed Manipulation Detection Code, CRYPTO 86 (1986), Santa Barbara, CA, 327-346.

[JUE89] Jueneman, R. R., Integrity Controls for Military and Commercial Applications, II, Invitational Workshop on Data Integrity, 1989.

[KAR87] Karger, P. A., Limiting the Damage Potential of Discretionary Trojan Horses, IEEE Conference on Security and Privacy 1987, p 32-37

[KAT73] Katzan, H., Computer Data Security, Van Nostrand Reinhold, Ltd., New York, 1973.

[KNU81] Knuth, D. E., The Art of Computer Programming, Volume 2, Seminumerical Algorithms, Addison-Wesley Publishing Company, Reading, MA, 1981.

[LEE88] Lee, T. M., Using Mandatory Integrity to Enforce "Commercial" Security, IEEE Conference on Security and Privacy 1988, 140-146.

[LIP82] Lipner, S. B., Non-Discretionary Controls for Commercial Applications, IEEE Security and Privacy 1982, 2-10.

[MAE87] Maekawa, M., Oldehoeft, A. E., Oldehoeft, R. R., Operating Systems Advanced Concepts, Benjamin/Cummings Publishing Company, Menlo Park, CA, 1987.

[MER87] Merkle, Ralph C., A Digital Signature Based on a Conventional Encryption Function, CRYPTO 87 (1987), Santa Barbara, CA, 369-378.

[MEY82] Meyer, C. H., Matyas, S. M., Cryptography: a new dimension in Computer Data Security, John Wiley & Sons, Inc. New York, 1982.

[MIZ87] Mizuno, M., Oldehoeft, A. E., An Access Control Language for Object-Oriented Programming Systems, Report TR-CS-87-12 Kansas State University, November 1987.

[NBS80] National Bureau of Standards, Federal Information Processing Standards Publication FIPS PUB 81, DES Modes of Operation, 1980.

[POZ86] Pozzo, M. M., Gray, T. E., An Approach to Containing Computer Viruses, Computers and Security 6 (4), August 1987, 321- 331.

[RIV78] Rivest, R.L., Shamir, A., Adleman, L., A Method for Obtaining Digital Signatures and Public Key Cryptosystems, Communications of the ACM, Feb 1978, V 21 #2, 120-126.

[SHO87] Shockly, W. R., Implementing the Clark/Wilson Integrity Policy using Current Technology, NCSC Computer Security Conference, 1987, 29-37.

[SIM85] Simmons, F. J., Authentication Theory/Coding Theory, CRYPTO 85 (1985), Santa Barbara, CA, 411-431.

[STI87] Stinson, D. R., A Construction for Authentication/Secrecy Codes from Certain Combinatorial Designs, CRYPTO 86 (1986), Santa Barbara, CA, 356-366.

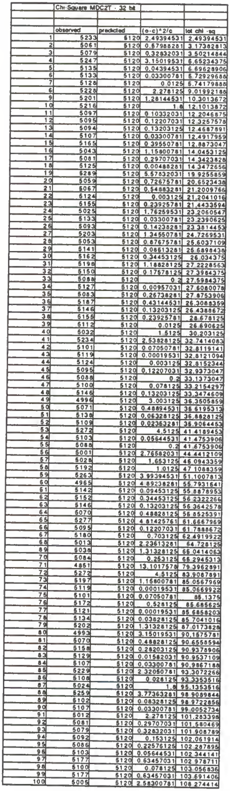
[TAN87] Tanenbaum, A. S., Operating Systems - Design and Implementation, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[TAN88] Tanenbaum, A. S., Computer Networks, Prentice-Hall, Englewood Cliffs, NJ, 1988.

Appendix - Chi-Square MDC2

		Chi-Square MDC2 - 32 bit		
	observed	predicted	(o-e)^2/e	tot chi -sq
1	5192	5120	1.0125	1.0125
1	5125	5120	0.00488281	1.01738281
1	5063	5120	0.63457031	1.65195313
1	5177	5120	0.63457031	2.28652344
1	5169	5120	0.46894531	2.75546875
1	5138	5120	0.06328125	2.81875
1	4993	5120	3.15019531	5.96894531
1	5228	5120	2.278125	8.24707031
9	5156	5120	0.253125	8.50019531
10	5164	5120	0.378125	8.87832031
11	5200	5120	1.25	10.1283203
12	5125	5120	0.00488281	10.1332031
13	4994	5120	3.10078125	13.2339844
14	5117	5120	0.00175781	13.2357422
15	4952	5120	5.5125	18.7482422
16	5184	5120	0.8	19.5482422
17	5096	5120	0.1125	19.6607422
18	5223	5120	2.07207031	21.7328125
19	5152	5120	0.2	21.9328125
20	4985	5120	3.55957031	25.4923828
21	5093	5120	0.14238281	25.6347656
22	5162	5120	0.34453125	25.9792969
23	5191	5120	0.98457031	26.9638672
24	5101	5120	0.07050781	27.034375
25	5220	5120	1.953125	28.9875
26	5055	5120	0.82519531	29.8126953
27	5174	5120	0.56953125	30.3822266
28	5065	5120	0.59082031	30.9730469
29	5266	5120	4.16328125	35.1363281
30	5191	5120	0.98457031	36.1208984
31	5211	5120	1.61738281	37.7382813
32	5140	5120	0.078125	37.8164063
33	5170	5120	0.48828125	38.3046875
34	5036	5120	1.378125	39.6828125
35	5089	5120	0.18769531	39.8705078
36	5080	5120	0.3125	40.1830078
37	5227	5120	2.23613281	42.4191406
38	5174	5120	0.56953125	42.9886719
39	5138	5120	0.06328125	43.0519531
40	5047	5120	1.04082031	44.0927734
41	5091	5120	0.16425781	44.2570313
42	5075	5120	0.39550781	44.6525391
43	5079	5120	0.32832031	44.9808594
44	5226	5120	2.19453125	47.1753906
45	5137	5120	0.05644531	47.2318359
46	5076	5120	0.378125	47.6099609
47	5077	5120	0.36113281	47.9710938
48	5138	5120	0.06328125	48.034375
49	5002	5120	2.71953125	50.7539063
50	5082	5120	0.28203125	51.0359375
51	5126	5120	0.00703125	51.0429688
52	5118	5120	0.00078125	51.04375
53	5261	5120	3.88300781	54.9267578
54	5056	5120	0.8	55.7267578
55	5062	5120	0.65703125	56.3837891
56	5065	5120	0.59082031	56.9746094
57	5028	5120	1.653125	58.6277344
58	5188	5120	0.903125	59.5308594
59	5088	5120	0.2	59.7308594
60	4961	5120	4.93769531	64.6685547
61	5232	5120	2.45	67.1185547
62	5124	5120	0.003125	67.1216797
63	5119	5120	0.00019531	67.121875
64	5250	5120	3.30078125	70.4226563
65	5214	5120	1.72578125	72.1484375
66	5045	5120	1.09863281	73.2470703
67	5155	5120	0.23925781	73.4863281
68	5024	5120	1.8	75.2863281
69	5033	5120	1.47832031	76.7646484
70	5005	5120	2.58300781	79.3476563
71	5011	5120	2.32050781	81.6681641
72	5218	5120	1.87578125	83.5439453
73	5114	5120	0.00703125	83.5509766
74	5135	5120	0.04394531	83.5949219
75	5151	5120	0.18769531	83.7826172
76	5066	5120	0.56953125	84.3521484
77	5119	5120	0.00019531	84.3523438
78	5129	5120	0.01582031	84.3681641
79	5064	5120	0.6125	84.9806641
80	5037	5120	1.34550781	86.3261719
81	5150	5120	0.17578125	86.5019531
82	5094	5120	0.13203125	86.6339844
83	5082	5120	0.28203125	86.9160156
84	5146	5120	0.13203125	87.0480469
85	5203	5120	1.34550781	88.3935547
86	5085	5120	0.23925781	88.6328125
87	5168	5120	0.45	89.0828125
88	5135	5120	0.04394531	89.1267578
89	5148	5120	0.153125	89.2798828
90	5173	5120	0.54863281	89.8285156
91	5043	5120	1.15800781	90.9865234
92	5131	5120	0.02363281	91.0101563
93	5207	5120	1.47832031	92.4884766
94	5146	5120	0.13203125	92.6205078
95	5016	5120	2.1125	94.7330078
96	5166	5120	0.41328125	95.1462891
97	5168	5120	0.45	95.5962891
98	5173	5120	0.54863281	96.1449219
99	5168	5120	0.45	96.5949219
100	4954	5120	5.38203125	101.976953

Appendix - Chi-Square MDC2T



	Chi-Square MDC2T - 32 bit			
	observed	predicted	(o-c)^2/c	tot chi -sq
1	5233	5120	2.49394531	2.49394531
2	5061	5120	0.67988281	3.17382813
3	5079	5120	0.32832031	3.50214844
4	5247	5120	3.15019531	6.65234375
5	5135	5120	0.04394531	6.69628906
6	5133	5120	0.03300781	6.72929688
7	5128	5120	0.0125	6.74179688
8	5228	5120	2.278125	9.01992188
9	5201	5120	1.28144531	10.3013672
10	5216	5120	1.8	12.1013672
11	5097	5120	0.10332031	12.2046875
12	5095	5120	0.12207031	12.3267578
13	5094	5120	0.13203125	12.4587891
14	5107	5120	0.03300781	12.4917969
15	5165	5120	0.39550781	12.8873047
16	5043	5120	1.15800781	14.0453125
17	5081	5120	0.29707031	14.3423828
18	5125	5120	0.00488281	14.3472656
19	5289	5120	5.57832031	19.9255859
20	5059	5120	0.72675781	20.6523438
21	5067	5120	0.54863281	21.2009766
22	5124	5120	0.003125	21.2041016
23	5155	5120	0.23925781	21.4433594
24	5025	5120	1.76269531	23.2060547
25	5133	5120	0.03300781	23.2390625
26	5093	5120	0.14238281	23.3814453
27	5203	5120	1.34550781	24.7269531
28	5053	5120	0.87675781	25.6037109
29	5141	5120	0.08613281	25.6898438
30	5162	5120	0.34453125	26.034375
31	5198	5120	1.18828125	27.2226563
32	5150	5120	0.17578125	27.3984375
33	5088	5120	0.2	27.5984375
34	5127	5120	0.00957031	27.6080078
35	5083	5120	0.26738281	27.8753906
36	5167	5120	0.43144531	28.3068359
37	5146	5120	0.13203125	28.4388672
38	5155	5120	0.23925781	28.678125
39	5112	5120	0.0125	28.690625
40	5032	5120	1.5125	30.203125
41	5234	5120	2.53828125	32.7414063
42	5101	5120	0.07050781	32.8119141
43	5119	5120	0.00019531	32.8121094
44	5124	5120	0.003125	32.8152344
45	5095	5120	0.12207031	32.9373047
46	5088	5120	0.2	33.1373047
47	5100	5120	0.078125	33.2154297
48	5146	5120	0.13203125	33.3474609
49	4996	5120	3.003125	36.3505859
50	5071	5120	0.46894531	36.8195313
51	5138	5120	0.06328125	36.8828125
52	5109	5120	0.02363281	36.9064453
53	5272	5120	4.5125	41.4189453
54	5103	5120	0.05644531	41.4753906
55	5088	5120	0.2	41.6753906
56	5001	5120	2.76582031	44.4412109
57	5028	5120	1.653125	46.0943359
58	5192	5120	1.0125	47.1068359
59	5263	5120	3.99394531	51.1007813
60	4965	5120	4.69238281	55.7931641
61	5142	5120	0.09453125	55.8876953
62	5162	5120	0.34453125	56.2322266
63	5146	5120	0.13203125	56.3642578
64	5070	5120	0.48828125	56.8525391
65	5277	5120	4.81425781	61.6667969
66	5095	5120	0.12207031	61.7888672
67	5180	5120	0.703125	62.4919922
68	5013	5120	2.23613281	64.728125
69	5038	5120	1.31328125	66.0414063
70	5084	5120	0.253125	66.2945313
71	4861	5120	13.1017578	79.3962891
72	5272	5120	4.5125	83.9087891
73	5197	5120	1.15800781	85.0667969
74	5119	5120	0.00019531	85.0669922
75	5101	5120	0.07050781	85.1375
76	5172	5120	0.528125	85.665625
77	5121	5120	0.00019531	85.6658203
78	5134	5120	0.03828125	85.7041016
79	5202	5120	1.31328125	87.0173828
80	4993	5120	3.15019531	90.1675781
81	5070	5120	0.48828125	90.6558594
82	5158	5120	0.28203125	90.9378906
83	5129	5120	0.01582031	90.9537109
84	5107	5120	0.03300781	90.9867188
85	5229	5120	2.32050781	93.3072266
86	5108	5120	0.028125	93.3353516
87	5024	5120	1.8	95.1353516
88	5259	5120	3.77363281	98.9089844
89	5102	5120	0.06328125	98.9722656
90	5107	5120	0.03300781	99.0052734
91	5012	5120	2.278125	101.283398
92	5081	5120	0.29707031	101.580469
93	5079	5120	0.32832031	101.908789
94	5092	5120	0.153125	102.061914
95	5086	5120	0.22578125	102.287695
96	5103	5120	0.05644531	102.344141
97	5177	5120	0.63457031	102.978711
98	5100	5120	0.078125	103.056836
99	5177	5120	0.63457031	103.691406
100	5005	5120	2.58300781	106.274414

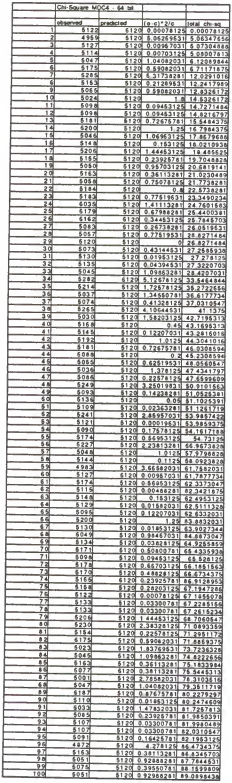
Appendix - Chi-Square MDC4 (32 bit checksum length)

	Chi Square MDC4 - 32 bit			
	observed	predicted	(o-c)^2/c	total chi sq
1	6049	5120	168.562695	168.562695
2	5728	5120	72.2	240.762695
3	5859	5120	106.664258	347.426953
4	5905	5120	120.356445	467.783398
5	5880	5120	112.8125	580.595898
6	5599	5120	44.8126953	625.408594
7	5510	5120	29.7070313	655.115625
8	5540	5120	34.453125	689.56875
9	4972	5120	4.278125	693.846875
10	5057	5120	0.77519531	694.62207
11	5083	5120	0.26738281	694.889453
12	5150	5120	0.17578125	695.065234
13	5043	5120	1.15800781	696.223242
14	5226	5120	2.19453125	698.417773
15	5075	5120	0.39550781	698.813281
16	5128	5120	0.0125	698.825781
17	5200	5120	1.25	700.075781
18	5002	5120	2.71953125	702.795313
19	5122	5120	0.00078125	702.796094
20	5154	5120	0.22578125	703.021875
21	5237	5120	2.67363281	705.695508
22	5107	5120	0.03300781	705.728516
23	5110	5120	0.01953125	705.748047
24	5235	5120	2.58300781	708.331055
25	4945	5120	5.98144531	714.3125
26	5316	5120	7.503125	721.815625
27	5105	5120	0.04394531	721.85957
28	5167	5120	0.43144531	722.291016
29	5113	5120	0.00957031	722.300586
30	5129	5120	0.01582031	722.316406
31	5186	5120	0.85078125	723.167188
32	5085	5120	0.23925781	723.406445
33	5047	5120	1.04082031	724.447266
34	5071	5120	0.46894531	724.916211
35	5212	5120	1.653125	726.569336
36	5286	5120	5.38203125	731.951367
37	5158	5120	0.28203125	732.233398
38	4999	5120	2.85957031	735.092969
39	5039	5120	1.28144531	736.374414
40	5118	5120	0.00078125	736.375195
41	5148	5120	0.153125	736.52832
42	5149	5120	0.16425781	736.692578
43	5130	5120	0.01953125	736.712109
44	5083	5120	0.26738281	736.979492
45	5093	5120	0.14238281	737.121875
46	5157	5120	0.26738281	737.389258
47	5075	5120	0.39550781	737.784766
48	5135	5120	0.04394531	737.828711
49	4962	5120	4.87578125	742.704492
50	5061	5120	0.67988281	743.384375
51	5139	5120	0.07050781	743.454883
52	5088	5120	0.2	743.654883
53	5300	5120	6.328125	749.983008
54	5157	5120	0.26738281	750.250391
55	5101	5120	0.07050781	750.320898
56	4993	5120	3.15019531	753.471094
57	5141	5120	0.08613281	753.557227
58	5008	5120	2.45	756.007227
59	5139	5120	0.07050781	756.077734
60	5181	5120	0.72675781	756.804492
61	5109	5120	0.02363281	756.828125
62	5158	5120	0.28203125	757.110156
63	5018	5120	2.03203125	759.142188
64	5192	5120	1.0125	760.154688
65	4976	5120	4.05	764.204688
66	5284	5120	5.253125	769.457813
67	5082	5120	0.28203125	769.739844
68	5102	5120	0.06328125	769.803125
69	5264	5120	4.05	773.853125
70	5014	5120	2.19453125	776.047656
71	5137	5120	0.05644531	776.104102
72	5067	5120	0.54863281	776.652734
73	5164	5120	0.378125	777.030859
74	5044	5120	1.128125	778.158984
75	5159	5120	0.29707031	778.456055
76	5175	5120	0.59082031	779.046875
77	5067	5120	0.54863281	779.595508
78	5053	5120	0.87675781	780.472266
79	5103	5120	0.05644531	780.528711
80	5265	5120	4.10644531	784.635156
81	5181	5120	0.72675781	785.361914
82	5163	5120	0.36113281	785.723047
83	5089	5120	0.18769531	785.910742
84	5154	5120	0.22578125	786.136523
85	5112	5120	0.0125	786.149023
86	5145	5120	0.12207031	786.271094
87	5115	5120	0.00488281	786.275977
88	5141	5120	0.08613281	786.362109
89	5150	5120	0.17578125	786.537891
90	5157	5120	0.26738281	786.805273
91	5633	5120	51.4001953	838.205469
92	5820	5120	95.703125	933.908594
93	5771	5120	82.7736328	1016.68223
94	5534	5120	33.4757813	1050.15801
95	5634	5120	51.6007813	1101.75879
96	5947	5120	133.579883	1235.33867
97	5798	5120	89.7820313	1325.1207
98	5693	5120	64.1267578	1389.24746
99	143	5120	4837.99395	6227.24141
100	0	5120	5120	11347.2414

Appendix - Chi-Square MDC4T (32 bit checksum length)

	Chi-Square MDC4T - 32 bit		
observed	predicted	(e-c)^2/c	total chi-sq
6101	5120	187.961133	187.961133
5873	5120	110.743945	298.705078
5642	5120	53.2195313	351.924609
5870	5120	109.863281	461.787891
5915	5120	123.442383	585.230273
5525	5120	32.0361328	617.266406
5676	5120	60.378125	677.644531
5543	5120	34.9470703	712.591602
5075	5120	0.39550781	712.987109
5165	5120	0.39550781	713.382617
5146	5120	0.13203125	713.514648
5157	5120	0.26738281	713.782031
5131	5120	0.02363281	713.805664
5113	5120	0.00957031	713.815234
5061	5120	0.67988281	714.495117
5165	5120	0.39550781	714.890625
5125	5120	0.00488281	714.895508
5187	5120	0.87675781	715.772266
5207	5120	1.47832031	717.250586
5173	5120	0.54863281	717.799219
5115	5120	0.00488281	717.804102
5153	5120	0.21269531	718.016797
5175	5120	0.59082031	718.607617
5106	5120	0.03828125	718.645898
5032	5120	1.5125	720.158398
5132	5120	0.028125	720.186523
5037	5120	1.34550781	721.532031
5132	5120	0.028125	721.560156
5129	5120	0.01582031	721.575977
5160	5120	0.3125	721.888477
4963	5120	4.81425781	726.702734
5106	5120	0.03828125	726.741016
5007	5120	2.49394531	729.234961
5110	5120	0.01953125	729.254492
5194	5120	1.06953125	730.324023
5066	5120	0.56953125	730.893555
5184	5120	0.8	731.693555
5143	5120	0.10332031	731.796875
5311	5120	7.12519531	738.92207
5155	5120	0.23925781	739.161328
5118	5120	0.00078125	739.162109
5184	5120	0.8	739.962109
4986	5120	3.50703125	743.469141
5119	5120	0.00019531	743.469336
5112	5120	0.0125	743.481836
5134	5120	0.03828125	743.520117
5168	5120	0.45	743.970117
5096	5120	0.1125	744.082617
5157	5120	0.26738281	744.35
5001	5120	2.76582031	747.11582
5106	5120	0.03828125	747.154102
5156	5120	0.253125	747.407227
4943	5120	6.11894531	753.526172
5026	5120	1.72578125	755.251953
5098	5120	0.09453125	755.346484
5100	5120	0.078125	755.424609
5144	5120	0.1125	755.537109
5036	5120	1.378125	756.915234
5046	5120	1.06953125	757.984766
5118	5120	0.00078125	757.985547
5073	5120	0.43144531	758.416992
5216	5120	1.8	760.216992
5085	5120	0.23925781	760.45625
5188	5120	0.903125	761.359375
4960	5120	5	766.359375
5095	5120	0.12207031	766.481445
5160	5120	0.3125	766.793945
5130	5120	0.01953125	766.813477
5136	5120	0.05	766.863477
5196	5120	1.128125	767.991602
5059	5120	0.72675781	768.718359
5229	5120	2.32050781	771.038867
5019	5120	1.99238281	773.03125
5067	5120	0.54863281	773.579883
4986	5120	3.50703125	777.086914
5116	5120	0.003125	777.090039
5147	5120	0.14238281	777.232422
5101	5120	0.07050781	777.30293
5095	5120	0.12207031	777.425
5119	5120	0.00019531	777.425195
5116	5120	0.003125	777.42832
5112	5120	0.0125	777.44082
5009	5120	2.40644531	779.847266
5161	5120	0.32832031	780.175586
5275	5120	4.69238281	784.867969
5091	5120	0.16425781	785.032227
5123	5120	0.00175781	785.033984
5144	5120	0.1125	785.146484
5314	5120	7.35078125	792.497266
5065	5120	0.59082031	793.088086
5537	5120	33.9626953	827.050781
5754	5120	78.5070313	905.557813
5796	5120	89.253125	994.810938
5665	5120	58.0126953	1052.82363
5828	5120	97.903125	1150.72676
6005	5120	152.973633	1303.70039
5865	5120	108.40332	1412.10371
5703	5120	66.3845703	1478.48828
152	5120	4820.5125	6299.00078
0	5120	5120	11419.0008

Appendix - Chi-Square MDC4 (64 bit checksum length)



	Chi-Square MDC4 - 64 bit			
	observed	predicted	(o-c)^2/c	total chi-sq
1	5122	5120	0.00078125	0.00078125
2	4959	5120	5.06269531	5.06347656
3	5127	5120	0.00957031	5.07304688
4	5114	5120	0.00703125	5.08007813
5	5047	5120	1.04082031	6.12089844
6	5175	5120	0.59082031	6.71171875
7	5285	5120	5.31738281	12.0291016
8	5153	5120	0.21269531	12.2417969
9	5065	5120	0.59082031	12.8326172
10	5024	5120	1.8	14.6326172
11	5098	5120	0.09453125	14.7271484
12	5098	5120	0.09453125	14.8216797
13	5181	5120	0.72675781	15.5484375
14	5200	5120	1.25	16.7984375
15	5046	5120	1.06953125	17.8679688
16	5148	5120	0.153125	18.0210938
17	5206	5120	1.44453125	19.465625
18	5155	5120	0.23925781	19.7048828
19	5050	5120	0.95703125	20.6619141
20	5163	5120	0.36113281	21.0230469
21	5058	5120	0.75078125	21.7738281
22	5184	5120	0.8	22.5738281
23	5183	5120	0.77519531	23.3490234
24	5035	5120	1.41113281	24.7601563
25	5179	5120	0.67988281	25.4400391
26	5162	5120	0.34453125	25.7845703
27	5083	5120	0.26738281	26.0519531
28	5057	5120	0.77519531	26.8271484
29	5120	5120	0	26.8271484
30	5073	5120	0.43144531	27.2585938
31	5130	5120	0.01953125	27.278125
32	5135	5120	0.04394531	27.3220703
33	5045	5120	1.09863281	28.4207031
34	5282	5120	5.12578125	33.5464844
35	5214	5120	1.72578125	35.2722656
36	5037	5120	1.34550781	36.6177734
37	5074	5120	0.41328125	37.0310547
38	5265	5120	4.10644531	41.1375
39	5030	5120	1.58203125	42.7195313
40	5168	5120	0.45	43.1695313
41	5145	5120	0.12207031	43.2916016
42	5192	5120	1.0125	44.3041016
43	5181	5120	0.72675781	45.0308594
44	5088	5120	0.2	45.2308594
45	5055	5120	0.82519531	46.0560547
46	5036	5120	1.378125	47.4341797
47	5086	5120	0.22578125	47.6599609
48	5249	5120	3.25019531	50.9101563
49	5093	5120	0.14238281	51.0525391
50	5136	5120	0.05	51.1025391
51	5109	5120	0.02363281	51.1261719
52	5241	5120	2.85957031	53.9857422
53	5121	5120	0.00019531	53.9859375
54	5090	5120	0.17578125	54.1617188
55	5174	5120	0.56953125	54.73125
56	5227	5120	2.23613281	56.9673828
57	5048	5120	1.0125	57.9798828
58	5144	5120	0.1125	58.0923828
59	4983	5120	3.66582031	61.7582031
60	5127	5120	0.00957031	61.7677734
61	5174	5120	0.56953125	62.3373047
62	5115	5120	0.00488281	62.3421875
63	5148	5120	0.153125	62.4953125
64	5129	5120	0.01582031	62.5111328
65	5095	5120	0.12207031	62.6332031
66	5200	5120	1.25	63.8832031
67	5130	5120	0.01953125	63.9027344
68	5049	5120	0.98457031	64.8873047
69	5134	5120	0.03828125	64.9255859
70	5171	5120	0.50800781	65.4335938
71	5098	5120	0.09453125	65.528125
72	5178	5120	0.65703125	66.1851563
73	5170	5120	0.48828125	66.6734375
74	5155	5120	0.23925781	66.9126953
75	5158	5120	0.28203125	67.1947266
76	5122	5120	0.00078125	67.1955078
77	5133	5120	0.03300781	67.2285156
78	5133	5120	0.03300781	67.2615234
79	5206	5120	1.44453125	68.7060547
80	5230	5120	2.36328125	71.0693359
81	5154	5120	0.22578125	71.2951172
82	5175	5120	0.59082031	71.8859375
83	5023	5120	1.83769531	73.7236328
84	5045	5120	1.09863281	74.8222656
85	5163	5120	0.36113281	75.1833984
86	5077	5120	0.36113281	75.5445313
87	5001	5120	2.76582031	78.3103516
88	5047	5120	1.04082031	79.3511719
89	5187	5120	0.87675781	80.2279297
90	5110	5120	0.01953125	80.2474609
91	5033	5120	1.47832031	81.7257813
92	5085	5120	0.23925781	81.9650391
93	5107	5120	0.03300781	81.9980469
94	5107	5120	0.03300781	82.0310547
95	5091	5120	0.16425781	82.1953125
96	4972	5120	4.278125	86.4734375
97	5163	5120	0.36113281	86.8345703
98	5051	5120	0.92988281	87.7644531
99	5075	5120	0.39550781	88.1599609
100	5051	5120	0.92988281	89.0898438

Appendix - Chi-Square MDC4T (64 bit checksum length)

	Chi-Square MDC4T - 64 bit			
	observed	predicted	(o-c)^2/c	total chi sq
1	5114	5120	0.00703125	0.00703125
2	5252	5120	3.403125	3.41015625
3	5019	5120	1.99238281	5.40253906
4	5126	5120	0.00703125	5.40957031
5	5237	5120	2.67363281	8.08320313
6	5080	5120	0.3125	8.39570313
7	4994	5120	3.10078125	11.4964844
8	5156	5120	0.253125	11.7496094
9	5141	5120	0.08613281	11.8357422
10	5202	5120	1.31328125	13.1490234
11	5021	5120	1.91425781	15.0632813
12	4881	5120	11.1564453	26.2197266
13	5218	5120	1.87578125	28.0955078
14	5086	5120	0.22578125	28.3212891
15	5147	5120	0.14238281	28.4636719
16	5098	5120	0.09453125	28.5582031
17	5106	5120	0.03828125	28.5964844
18	5222	5120	2.03203125	30.6285156
19	5201	5120	1.28144531	31.9099609
20	5139	5120	0.07050781	31.9804688
21	4889	5120	10.4220703	42.4025391
22	5256	5120	3.6125	46.0150391
23	5221	5120	1.99238281	48.0074219
24	5107	5120	0.03300781	48.0404297
25	5193	5120	1.04082031	49.08125
26	5101	5120	0.07050781	49.1517578
27	5116	5120	0.003125	49.1548828
28	5128	5120	0.0125	49.1673828
29	5232	5120	2.45	51.6173828
30	5190	5120	0.95703125	52.5744141
31	5101	5120	0.07050781	52.6449219
32	5264	5120	4.05	56.6949219
33	5079	5120	0.32832031	57.0232422
34	5068	5120	0.528125	57.5513672
35	5210	5120	1.58203125	59.1333984
36	5155	5120	0.23925781	59.3726563
37	5214	5120	1.72578125	61.0984375
38	5098	5120	0.09453125	61.1929688
39	5165	5120	0.39550781	61.5884766
40	5034	5120	1.44453125	63.0330078
41	5125	5120	0.00488281	63.0378906
42	5109	5120	0.02363281	63.0615234
43	5191	5120	0.98457031	64.0460938
44	5281	5120	5.06269531	69.1087891
45	5031	5120	1.54707031	70.6558594
46	5042	5120	1.18828125	71.8441406
47	5152	5120	0.2	72.0441406
48	5168	5120	0.45	72.4941406
49	5257	5120	3.66582031	76.1599609
50	5144	5120	0.1125	76.2724609
51	5058	5120	0.75078125	77.0232422
52	5231	5120	2.40644531	79.4296875
53	5146	5120	0.13203125	79.5617188
54	5150	5120	0.17578125	79.7375
55	5099	5120	0.08613281	79.8236328
56	5208	5120	1.5125	81.3361328
57	5114	5120	0.00703125	81.3431641
58	5173	5120	0.54863281	81.8917969
59	5095	5120	0.12207031	82.0138672
60	5086	5120	0.22578125	82.2396484
61	5193	5120	1.04082031	83.2804688
62	5150	5120	0.17578125	83.45625
63	5062	5120	0.65703125	84.1132813
64	5165	5120	0.39550781	84.5087891
65	5159	5120	0.29707031	84.8058594
66	5067	5120	0.54863281	85.3544922
67	5037	5120	1.34550781	86.7
68	5053	5120	0.87675781	87.5767578
69	5092	5120	0.153125	87.7298828
70	5156	5120	0.253125	87.9830078
71	5069	5120	0.50800781	88.4910156
72	5037	5120	1.34550781	89.8365234
73	5152	5120	0.2	90.0365234
74	5038	5120	1.31328125	91.3498047
75	5054	5120	0.85078125	92.2005859
76	5096	5120	0.1125	92.3130859
77	5263	5120	3.99394531	96.3070313
78	5022	5120	1.87578125	98.1828125
79	5083	5120	0.26738281	98.4501953
80	5102	5120	0.06328125	98.5134766
81	5053	5120	0.87675781	99.3902344
82	5117	5120	0.00175781	99.3919922
83	5184	5120	0.8	100.191992
84	5132	5120	0.028125	100.220117
85	5005	5120	2.58300781	102.803125
86	5205	5120	1.41113281	104.214258
87	5096	5120	0.1125	104.326758
88	5045	5120	1.09863281	105.425391
89	5046	5120	1.06953125	106.494922
90	5143	5120	0.10332031	106.598242
91	5137	5120	0.05644531	106.654688
92	5070	5120	0.48828125	107.142969
93	5199	5120	1.21894531	108.361914
94	5026	5120	1.72578125	110.087695
95	5073	5120	0.43144531	110.519141
96	5165	5120	0.39550781	110.914648
97	5122	5120	0.00078125	110.91543
98	5002	5120	2.71953125	113.634961
99	5048	5120	1.0125	114.647461
100	4961	5120	4.93769531	119.585156

Appendix - Binomial Test MDC2

	Binomial Distribution of MDC2			
	observed	calculated	(o-c)^2/c	chi-square
0	0	0.0001	0.0001	0.0001
1	0	0.0038	0.0038	0.0039
2	0	0.059	0.059	0.0629
3	1	0.59	0.28491525	0.34781525
4	10	4.29	7.60002331	7.94783856
5	21	24	0.375	8.32283856
6	115	108.03	0.44969823	8.7725368
7	395	401.24	0.09704317	8.86957996
8	1295	1253.88	1.34849778	10.2180777
9	3377	3343.68	0.33203608	10.5501138
10	7697	7690.5	0.00549379	10.5556076
11	15161	15380.92	3.14446772	13.7000753
12	26765	26916.61	0.85395568	14.554031
13	41343	41410.16	0.10892171	14.6629527
14	56603	56199.51	2.89689679	17.5598495
15	67669	67439.4	0.78168192	18.3415314
16	71725	71654.37	0.06962027	18.4111517
17	67558	67439.4	0.20857184	18.6197236
18	56363	56199.51	0.47560877	19.0953323
19	41362	41410.16	0.05601006	19.1513424
20	26713	26916.61	1.54020258	20.691545
21	15243	15380.92	1.23672228	21.9282673
22	7563	7690.5	2.11380925	24.0420765
23	3311	3343.68	0.31940329	24.3614798
24	1221	1253.88	0.86219925	25.223679
25	369	401.24	2.59051341	27.8141925
26	96	108.03	1.33963621	29.1538287
27	22	24	0.16666667	29.3204953
28	2	4.29	1.22240093	30.5428963
29	0	0.59	0.59	31.1328963
30	0	0.06	0.06	31.1928963
31	0	0.003	0.003	31.1958963
32	0	0.0001	0.0001	31.1959963
	observed	calculated	(o-c)^2/c	chi-square

Appendix - Binomial Test MDC2T

	Binomial Distribution of MDC2T			
	observed	calculated	(o-c)^2/c	chi-square
0	0	0.0001	0.0001	0.0001
1	0	0.0038	0.0038	0.0039
2	0	0.059	0.059	0.0629
3	1	0.59	0.28491525	0.34781525
4	9	4.29	5.17111888	5.51893414
5	27	24	0.375	5.89393414
6	116	108.03	0.58799315	6.48192729
7	432	401.24	2.35813379	8.84006107
8	1223	1253.88	0.76049893	9.60056
9	3380	3343.68	0.39451814	9.99507814
10	7779	7690.5	1.01843183	11.01351
11	15227	15380.92	1.5403088	12.5538188
12	26851	26916.61	0.15992624	12.713745
13	41345	41410.16	0.10253101	12.816276
14	56424	56199.51	0.89672953	13.7130055
15	67649	67439.4	0.65143166	14.3644372
16	71655	71654.37	5.5391E-06	14.3644427
17	67669	67439.4	0.78168192	15.1461247
18	56075	56199.51	0.27585187	15.4219765
19	41549	41410.16	0.4655028	15.8874793
20	26671	26916.61	2.24115415	18.1286335
21	15355	15380.92	0.04368051	18.172314
22	7546	7690.5	2.71507054	20.8873845
23	3278	3343.68	1.29015408	22.1775386
24	1217	1253.88	1.08474049	23.2622791
25	399	401.24	0.01250523	23.2747843
26	99	108.03	0.75479867	24.029583
27	22	24	0.16666667	24.1962497
28	2	4.29	1.22240093	25.4186506
29	0	0.59	0.59	26.0086506
30	0	0.06	0.06	26.0686506
31	0	0.003	0.003	26.0716506
32	0	0.0001	0.0001	26.0717506
	observed	calculated	(o-c)^2/c	chi-square

Appendix - Binomial Test MDC4 (32 bit checksum length)

	Binomial Distribution of MDC4			
	observed	calculated	(o-c)^2/c	chi-square
0	0	0.0001	0.0001	0.0001
1	0	0.0038	0.0038	0.0039
2	0	0.059	0.059	0.0629
3	1	0.59	0.28491525	0.34781525
4	10	4.29	7.60002331	7.94783856
5	66	24	73.5	81.4478386
6	200	108.03	78.2975183	159.745357
7	729	401.24	267.736561	427.481918
8	1993	1253.88	435.686329	863.168247
9	4912	3343.68	735.604969	1598.77322
10	10507	7690.5	1031.48979	2630.26301
11	20073	15380.92	1431.35877	4061.62178
12	32861	26916.61	1312.78688	5374.40866
13	48784	41410.16	1313.04772	6687.45638
14	63562	56199.51	964.532591	7651.98897
15	72501	67439.4	379.893572	8031.88255
16	73673	71654.37	56.8683679	8088.75091
17	64639	67439.4	116.285734	8205.03665
18	49865	56199.51	713.992292	8919.02894
19	33459	41410.16	1526.70131	10445.7302
20	19362	26916.61	2120.33136	12566.0616
21	9284	15380.92	2416.78869	14982.8503
22	3792	7690.5	1976.24371	16959.094
23	1343	3343.68	1197.10034	18156.1943
24	305	1253.88	718.069715	18874.2641
25	61	401.24	288.513751	19162.7778
26	18	108.03	75.0291669	19237.807
27	0	24	24	19261.807
28	0	4.29	4.29	19266.097
29	0	0.59	0.59	19266.687
30	0	0.06	0.06	19266.747
31	0	0.003	0.003	19266.75
32	0	0.0001	0.0001	19266.7501
	observed	calculated	(o-c)^2/c	chi-square

Appendix - Binomial Test MDC4T (32 bit checksum length)

	Binomial Distribution of MDC4T			
	observed	calculated	(o-c)^2/c	chi-square
0	0	0.0001	0.0001	0.0001
1	0	0.0038	0.0038	0.0039
2	2	0.059	63.8556102	63.8595102
3	3	0.59	9.84423729	73.7037475
4	8	4.29	3.20841492	76.9121624
5	59	24	51.0416667	127.953829
6	190	108.03	62.1964352	190.150264
7	670	401.24	180.021776	370.172041
8	1901	1253.88	333.974778	704.146819
9	4784	3343.68	620.430694	1324.57751
10	10347	7690.5	917.624634	2242.20215
11	20058	15380.92	1422.22164	3664.42379
12	33126	26916.61	1432.44354	5096.86733
13	49214	41410.16	1470.65162	6567.51895
14	63374	56199.51	915.903124	7483.42207
15	72883	67439.4	439.398645	7922.82072
16	73417	71654.37	43.3590375	7966.17976
17	64515	67439.4	126.811854	8092.99161
18	50503	56199.51	577.411194	8670.4028
19	33198	41410.16	1628.5755	10298.9783
20	18870	26916.61	2405.50101	12704.4793
21	9358	15380.92	2358.47825	15062.9576
22	3851	7690.5	1916.8793	16979.8369
23	1238	3343.68	1326.05042	18305.8873
24	348	1253.88	654.463405	18960.3507
25	72	401.24	270.159948	19230.5106
26	10	108.03	88.9556688	19319.4663
27	1	24	22.0416667	19341.508
28	0	4.29	4.29	19345.798
29	0	0.59	0.59	19346.388
30	0	0.06	0.06	19346.448
31	0	0.003	0.003	19346.451
32	0	0.0001	0.0001	19346.4511
	observed	calculated	(o-c)^2/c	chi-square

Appendix - Binomial Test MDC4 (64 bit checksum length)

	Binomial Distribution of MDC4 - 64 bit			
	observed	calculated	(o-c)^2/c	chi-square
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	0	0	0	0
10	0	0.004	0.004	0.004
11	0	0.02	0.02	0.024
12	0	0.09	0.09	0.114
13	0	0.36	0.36	0.474
14	2	1.33	0.3375188	0.8115188
15	6	4.43	0.55641084	1.36792963
18	7	13.56	3.17356932	4.54149895
17	46	38.3	1.54804178	6.08954073
18	106	99.97	0.38371812	6.45325884
19	238	242	0.0681157	8.51937455
20	552	544.56	0.1016483	8.62102285
21	1129	1140.98	0.12578696	6.74880981
22	2241	2230.09	0.05337368	6.80018348
23	3983	4072.34	2.93571647	9.73589995
24	8811	8958.91	3.06022782	12.7961276
25	11091	11131.05	0.14410163	12.9402292
26	16795	16696.58	0.58014853	13.5203777
27	23569	23498.89	0.20917635	13.7295541
28	31342	31052.11	2.70629635	18.4358504
29	38815	38547.45	0.11837365	18.5542241
30	45371	44972.01	3.53982444	20.0940485
31	49454	49324.13	0.34194657	20.4359951
32	50712	50865.53	0.46340736	20.8994025
33	49416	49324.13	0.17111497	21.0705174
34	45196	44972.01	1.11561858	22.188134
35	38785	38547.45	1.22778556	23.4139196
36	30825	31052.11	1.661045	25.0749646
37	23290	23498.89	1.85689759	26.9318622
38	16437	16696.58	4.03566338	30.9675255
39	11075	11131.05	0.28223775	31.2497633
40	6765	6956.91	5.2939377	38.543701
41	3980	4072.34	2.09380248	38.6375035
42	2140	2230.09	3.83940832	42.2769118
43	1111	1140.98	0.78774422	43.064656
44	562	544.56	0.55853092	43.6231869
45	237	242	0.10330579	43.7264927
46	94	99.97	0.35851595	44.0830087
47	39	38.3	0.01279373	44.0958024
48	14	13.58	0.01427729	44.1100797
49	3	4.43	0.46160271	44.5716824
50	0	1.33	1.33	45.9016824
51	1	0.36	1.13777778	47.0394802
52	0	0.09	0.09	47.1294602
53	0	0.02	0.02	47.1494802
54	0	0.004	0.004	47.1534602
55	0	0	0	47.1534602
56	0	0	0	47.1534602
57	0	0	0	47.1534602
58	0	0	0	47.1534802
59	0	0	0	47.1534602
60	0	0	0	47.1534802
61	0	0	0	47.1534602
62	0	0	0	47.1534602
63	0	0	0	47.1534602
84	0	0	0	47.1534602

Appendix - Binomial Test MDC4T (64 bit checksum length)

		Binomial Distribution of MDC4T - 64 bit		
		observed	calculated	(o-c)^2/c
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	0	0	0	0
10	0	0.004	0.004	0.004
11	0	0.02	0.02	0.024
12	0	0.09	0.09	0.114
13	0	0.36	0.36	0.474
14	0	1.33	1.33	1.804
15	4	4.43	0.04173815	1.84573815
18	13	13.56	0.02312684	1.86886499
17	34	38.3	0.48276762	2.35163262
18	100	99.97	9.0027E-06	2.35184162
19	231	242	0.5	2.85164162
20	522	544.56	0.93461437	3.78625599
21	1154	1140.98	0.14857438	3.93483037
22	2238	2230.09	0.02805631	3.96288668
23	3999	4072.34	1.32080219	5.28368887
24	8986	8958.91	0.1216385	5.40532737
25	11188	11131.05	0.12265712	5.52798449
26	16762	16696.58	0.25632653	5.78431102
27	23733	23498.89	2.33234387	8.11665489
28	31182	31052.11	0.54332579	8.65998068
29	38914	38547.45	3.4855458	12.1455265
30	45219	44972.01	1.35648952	13.502016
31	49325	49324.13	1.5345E-05	13.5020313
32	50897	50865.53	0.01947018	13.5215015
33	49248	49324.13	0.11750389	13.6390054
34	45079	44972.01	0.25453299	13.8935384
35	38478	38547.45	0.12512637	14.0186648
36	30622	31052.11	5.95755368	19.9762184
37	23194	23498.89	3.95584268	23.9320611
38	16598	16698.58	0.58203635	24.5140975
39	11148	11131.05	0.02581091	24.5399084
40	6889	8956.91	0.66290487	25.202813
41	4039	4072.34	0.27295255	25.4757656
42	2238	2230.09	0.02805831	25.5038219
43	1087	1140.98	2.55380497	28.0576269
44	541	544.56	0.0232731	28.0809
45	216	242	2.79338843	30.8742884
46	94	99.97	0.35651595	31.2308044
47	25	38.3	4.81853786	35.8493422
48	19	13.56	2.18241888	38.0317811
49	3	4.43	0.46180271	38.4933638
50	1	1.33	0.0818797	38.5752435
51	0	0.36	0.36	38.9352435
52	0	0.09	0.09	39.0252435
53	0	0.02	0.02	39.0452435
54	0	0.004	0.004	39.0492435
55	0	0	0	39.0492435
56	0	0	0	39.0492435
57	0	0	0	39.0492435
58	0	0	0	39.0492435
59	0	0	0	39.0492435
60	0	0	0	39.0492435
81	0	0	0	39.0492435
82	0	0	0	39.0492435
83	0	0	0	39.0492435
84	0	0	0	39.0492435

Appendix - Program Code

```
/* MDC programs: MDC2, MDC2T, MDC4, MDC4T
   Only one of the programs run at a time, but structure
   available for all four.

   Generates checksums for 512,000 programs according to
   the algorithm hardcoded into the program.

   Note: if a different algorithm other than MDC4 is selected
         the assignmet to chksum must be changed.

*/

#include "rn.h"

long mrand48();

/* variables for data    */

long int a;       /* store for 32 bit random number */
long int b[2];            /* store for 16 bit blocks */
char c[4];        /* store for 8 bit blocks */
long int endline=0;    /* character to end program         */

/* variables and initial values for mdc equations
   (with and without tss)    */

long int m1a=16033,m2a=32707;
long int m1b=16033,m2b=32707;
long int m1c=229,m2c=113,m3c=227,m4c=127;
long int m1d=229,m2d=113,m3d=227,m4d=127;

FILE *fp1;        /* file handle for storage */

long int mod1(m,n)    /* for handling modular arithmatic */
long int m,n;
{
long int p;
p=m % n;
if (p<0) {
  p=p+n; }
return p;
};
```

Appendix - Program Code (cont)

```
void mdc32_2eq()       /* 32 bit 2 equation mdc  */
{

long int m1as,m2as;
long int m1ast,m2ast;

m1ast=(m1a^b[0]-m2a^b[1]);
m1as=modl(modl(m1ast,n1a)+modl(m1ast,n1a), n1a);

m2ast=(m2a^b[0]-m1as^b[1]);
m2as=modl(modl(m2ast,n2a)+modl(m2ast,n2a), n2a);

m1a=m1as;
m2a=m2as;

};

void mdc32_2eq_tss()    /* 32 bit 2 equation with tss
                     ( additional feed back term)   */
{

long int m1bs,m2bs;
long int m1bst,m2bst;
long tss;

tss=((b[0]&0xffff0000) | (b[1]&0xffff));

m1bst=(m1b^b[0]-m2b^b[1]);
m1bs=modl(modl(m1bst,n1b)+modl(m1bst,n1b), n1b);

m2bst=(m2b^b[0]-m1bs^b[1]);
m2bs=modl(modl(m2bst,n2b)+modl(m2bst,n2b), n2b);

m1b=m1bs;
m2b=m2bs;

};

void getrand()        /* function that gets the random number */
{
   a=mrand48();           /* get random number */
   b[0]=a & 0xffff;       /* split it into 16 bit chunks */
   b[1]=(a>>16) & 0xffff;
   t[0]=a & 0xff;                  /* split it into 8 bit chunks
   t[1]=(a>>8) & 0xff;
   t[2]=(a>>16) & 0xff;
   t[3]=(a>>24) & 0xff;
};
```

Appendix - Program Code (cont)

```
void mdc32_4eq_tss()      /* 32 bit 4 equation mdc
                        with tss (additional feedback) */
{

long int m1dst, m2dst,m3dst,m4dst;
long int m1ds, m2ds,m3ds,m4ds;
long int tss;

tss=((t[0]&0xf000) | (t[1]&0x0f00) |
   (t[2]&0x00f0) | (t[3]&0x000f));

m1dst=(m1d^t[1]-m2d^t[2]+m3d^t[3]-m4d^t[4]- tss);
m1ds=modl(modl(m1dst, n1d) + modl(m1dst, n1d), n1d);

m2dst=(m2d^t[1]-m3d^t[2]+m4d^t[3]-m1ds^t[4]- tss);
m2ds=modl(modl(m2dst, n2d) + modl(m2dst, n2d),  n2d);

m3dst=(m3d^t[1]-m4d^t[2]+m1ds^t[3]-m2ds^t[4]- tss);
m3ds=modl(modl(m3dst, n3d) + modl(m3dst, n3d),  n3d);

m4dst=(m4d^t[1]-m1ds^t[2]+m2ds^t[3]-m3ds^t[4]- tss);
m4ds=modl(modl(m4dst, n4d) + modl(m4dst, n4d), n4d);

m1d=m1ds;
m2d=m2ds;
m3d=m3ds;
m4d=m4ds;

};

void reinit()    /* reset values at start of new program */
{
   m1a=16033;m2a=32707;
   m1b=16033;m2b=32707;
   m1c=229;m2c=113;m3c=227;m4c=127;
   m1d=229;m2d=113;m3d=227;m4d=127;

};
```

Appendix - Program Code (cont)

```
void mdc32_4eq()        /* 32 bit 4 equation mdc  */
{

long int m1cst, m2cst,m3cst,m4cst,
long int m1cs, m2cs,m3cs,m4cs;

m1cst=(m1c^t[1]-m2c^t[2]+m3c^t[3]-m4c^t[4]);
m1cs=modl(modl(m1cst, n1c) + modl(m1cst, n1c), n1c);

m2cst=(m2c^t[1]-m3c^t[2]+m4c^t[3]-m1cs^t[4]);
m2cs=modl(modl(m2cst, n2c) + modl(m2cst, n2c),  n2c);

m3cst=(m3c^t[1]-m4c^t[2]+m1cs^t[3]-m2cs^t[4]);
m3cs=modl(modl(m3cst, n3c) + modl(m3cst, n3c),  n3c);

m4cst=(m4c^t[1]-m1cs^t[2]+m2cs^t[3]-m3cs^t[4]);
m4cs=modl(modl(m4cst, n4c) + modl(m4cst, n4c), n4c);

m1c=m1cs;
m2c=m2cs;
m3c=m3cs;
m4c=m4cs;

};
```

Appendix - Program Code (cont)

```
main()

{

int thiscnt=0, ncnt=200, cnt=0,cnt1=0;    /* counters to keep
                                          track of place in program

int cont;
unsigned int seed16v[3];
unsigned long chksum;
fd1=fopen("/usrc/varney/m32.4.t.data","w");

for (cnt=0;cnt<3;cnt++) seed16v[cnt]=0;    /* initialize rn seed */
cnt=0;
seed48(seed16v);    /* initialize rn generator */

cont=TRUE;
while (cont)
{


   getrand();    /* get the random program  */
   cnt++;

   if (b[0]==endline || thiscnt++==ncnt) { /* end of a prgm */
        ncnt=b[2];
        thiscnt=0;
        cnt1++;
        chksum=(m1c<<24)+(m2c<<16)+(m3c<<8)+m4c; /* make chksum */
        printf(fd1, "%x\t%d\t%d\n",chksum,cnt1,cnt);
        if (cnt1>512000) cont=FALSE;     /is it the last prgm? */
        if(b[0]==endline) {endline=b[1]; reinit();cnt=0; };
   }

   /*mdc32_2eq();*/       /* 32 bit 2 equation mdc  */

   /*mdc32_2eq_tss();*/         /* 32 bit 2 equation mdc  */

    mdc32_4eq();          /* 32 bit 4 equation mdc  */

    /* mdc32_4eq_tss();*/        /* 32 bit 4 equation mdc  */

 }
}
```

Construction, Testing and Use of Checksum Algorithms for Computer Virus Detection

by

Douglas William Varney

B.S., University of Virginia, 1980
M.B.A., University of Virginia, 1984

An Abstract of a Thesis

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

in

Department of Computer and Information Sciences

Kansas State University
Manhattan, Kansas

1989

ABSTRACT

This thesis deals with the construction and testing of checksum algorithms for computer virus detection on small computer systems. Checksum algorithms need to produce checksums with the following features: even mapping over the range of possible checksums, permutation dependency, and every bit of the checksum is an overdetermined function of all the bits of the set of data being checksummed. Checksum algorithms to protect against viruses also need to be noninvertable and produce checksums with adequate length because viruses can employ either a brute force or a trap door attack against the checksum. A birthday attack was shown to be not applicable in the case of strong checksum algorithms. The methods to construct checksum algorithms with these properties include substitution, transposition and feed back. Cryptographic checksum algorithms were found to be too inefficient for small computers and effort was concentrated on noncryptographic algorithms. Several noncryptographic checksum algorithms were created and shown to have the necessary features. These algorithms were also tested for efficiency (speed of execution). On the basis of the strength and efficiency of the checksum algorithms a recommendation of checksum algorithms for different types of small computers was presented.