Malicious software, computer viruses and some applications of distributed computing

Jeffrey Horton
SITACS, University of Wollongong, jeffh@uow.edu.au
Malicious Software, Computer Viruses and Some Applications of Distributed Computing

A thesis submitted in fulfillment of the requirements for the award of the degree

Doctor of Philosophy

from

UNIVERSITY OF WOLLONGONG

by

Jeffrey Horton

School of Information Technology and Computer Science

October 2000
Declaration

This is to certify that the work reported in this thesis was done by the author, unless specified otherwise, and that no part of it has been submitted in a thesis to any other university or similar institution.

Jeffrey Horton
October 12, 2000
Abstract

Computer viruses and other forms of malicious software have been a significant computer security problem for over ten years. We discuss computer viruses and the strategies that can be employed by a virus to complicate automated detection. In particular, the organisation of Macintosh operating system software and the strategies that have been employed by viruses on this platform are considered. Macro viruses, one of the relatively recent developments in the area of computer viruses, are described.

A previously unknown attack that could be implemented on a Macintosh and that would allow the infection of application programs without altering any executable code is described. We also discuss a simple countermeasure to this attack. We subsequently extend this attack and consider the effect of operating system changes on Macintosh viruses.

There are many other types of malicious software. Applets written in Java can be used to implement a variety of attacks against Internet users. We consider the potential of a Java applet to be used in conjunction with a Web Spoofing attack to perform covert distributed computing.

We continue by discussing two combinatorial search problems: finding sequences with zero autocorrelation function, and finding Hadamard matrices using Williamson's method. We report many new results for the search for sequences with zero autocorrelation function, and note that this search, if it is to be extended, would require the application of distributed computing techniques. We describe a distributed search for Hadamard matrices based on Williamson's method; although we do not find any new matrices, we provide independent verification of results presented by others.
Acknowledgements

My thanks to the following:

- My supervisor, Professor Jennifer Seberry, for her support, advice, encouragement and assistance;

- Dr. Christos Koukouvinos, for his careful reading and comments on the material in Chapter 7 and Chapter 8;

- My fellow students and the other academics at the Centre for Computer Security Research at the University of Wollongong, for providing a friendly working environment over the four years that have been required for the completion of this thesis;

- Camilla, who proofread.
List of Publications


5. Jeffrey Horton, Christos Koukouvinos and Jennifer Seberry. A Search for Hadamard Matrices constructed from Williamson Matrices. (submitted)
Contents

Abstract iv
Acknowledgements v
List of Publications vi
1 Introduction 1

I Computer Viruses and Other Forms of Malicious Software 4

2 Introduction to Computer Viruses 6
  2.1 Introduction ...................................................... 6
  2.2 A Little Bit of Theory ............................................ 10
  2.3 Platforms Affected .............................................. 10
  2.4 Computer Virus Types and Strategies ....................... 11
    2.4.1 IBM-PC Viruses ............................................. 12
      2.4.1.1 File Infecting Viruses ............................... 13
      2.4.1.2 Boot Sector Infecting Viruses ...................... 16
      2.4.1.3 Companion Viruses .................................... 19
      2.4.1.4 “Link” Viruses ....................................... 19
    2.4.2 Apple Macintosh Viruses ................................. 20
      2.4.2.1 Introduction .......................................... 20
      2.4.2.2 Macintosh Files ....................................... 21
      2.4.2.3 Resources ............................................. 22
      2.4.2.4 The Macintosh Finder and the “Desktop” ........... 25
4.4 Affect of Operating System Changes on Macintosh Viruses

5 Covert Distributed Computing Using Java Through Web Spoofing
5.1 Introduction
5.2 About Web Spoofing
5.2.1 Some HTML Tags to Modify
5.3 Application of Web Spoofing to Distributed Computing
5.3.1 An Implementation
5.4 Countermeasures?
5.4.1 Client-side Precautions
5.4.2 Server-side Precautions
5.5 Computing for Sale
5.6 Conclusion

II Some Applications of Distributed Computing to Combinatorial Search

6 Introduction to Sequences and Hadamard Matrices
6.1 Sequences
6.1.1 Introduction
6.1.2 Mathematical Background
6.1.3 Asymptotic Existence
6.2 Hadamard Matrices
6.2.1 Introduction
6.2.2 Basic Theory
6.2.3 Some constructions for Hadamard Matrices using “multiplication”
6.2.4 Direct construction theorems for Hadamard Matrices
6.2.5 Hadamard Matrices and Block Codes
6.2.6 Computer Searches for Hadamard Matrices of Williamson Type

7 When the necessary conditions are not sufficient: sequences with zero autocorrelation function
List of Tables

7.1 Variable counts for sequences $A$ and $B$ of type $(4,9)$. .......... 111
7.2 Variable counts for sequences $A$ and $B$ of type $(1,9)$. .......... 111
7.3 Time taken to search for sequences of types $(1,9)$, $(4,9)$ and $(1,16)$ for various $n$, multiple computers. ................. 112
7.4 The results of an exhaustive search for the existence of two sequences of types $(1,9)$, $(1,16)$ and $(4,9)$. ................. 113
7.5 Some new sequences. ............................................. 114
7.6 Asymptotic existence of $OD(2n;1,9)$, $OD(2n;1,16)$ and $OD(2n;4,9)$. 114
C.1 Hadamard matrices of orders 12, 20, 28, 36, 44, 52, 60 and 68 from Williamson Matrices ........................................... 171
C.2 Hadamard matrices of orders 76, 84 and 92 from Williamson Matrices 172
C.3 Hadamard matrices of orders 100 and 108 from Williamson Matrices . 173
C.4 Hadamard matrices of orders 116, 124 and 132 from Williamson Matrices . ................................................. 174
C.5 Hadamard matrices of orders 140 and 148 from Williamson Matrices . 175
C.6 Hadamard matrices from Williamson matrices for orders 12 through 60; alternative representation .................................. 176
C.7 Hadamard matrices from Williamson matrices for orders 68 and 76; alternative representation .................................. 177
C.8 Hadamard matrices from Williamson matrices for orders 84 and 92; alternative representation .................................. 178
C.9 Hadamard matrices from Williamson matrices for orders 100 and 108; alternative representation .................................. 179
C.10 Hadamard matrices from Williamson matrices for orders 116, 124 and 132; alternative representation .......................... 180
C.11 Hadamard matrices from Williamson matrices for orders 140 and 148; alternative representation .......................... 181
List of Figures

2.1 An overwriting virus. The start of the host program’s code is replaced with viral code. ......................................................... 14
2.2 An appending virus. The start of the host program’s code is replaced with a jump to appended viral code; the original start of the host is preserved. ......................................................... 15
2.3 A prepending virus. The host’s code is effectively appended to the viral code. ......................................................... 15
2.4 Simple illustration of boot process for floppy and hard disks. .... 17
2.5 Finder “desktop” display with some open windows. ............... 25
2.6 Simple illustration of a code segment based 68K application. The jump table (CODE 0) contains references to the entry points of routines that can be the subject of an intersegment reference. .......... 29
2.7 Illustration of modifications made by virus that adds an additional CODE resource to an application to achieve infection. ......... 33
2.8 Illustration of modifications made by virus that adds its code to an existing CODE resource and modifies a routine to jump to the viral code. ......................................................... 34
2.9 Illustration of modifications made by virus that adds its code to an existing CODE resource and modifies a jump table entry to refer to the viral code. ......................................................... 35
2.10 Illustration of typical relationship of application MENU and System MDEF. ......................................................... 35
2.11 Illustration of the relationship between an application MENU, a viral MDEF and the System MDEF. ......................................................... 36
2.12 Illustration of the relationship between an application Menu, a viral MDEF and the System MDEF.
2.13 Illustration of the relationship between an uninfected application and a System infected with a virus using an MDEF resource.
2.14 Illustration of the strategy employed by the WDEF virus: a WDEF resource located in a floppy disk’s Desktop file overrides the System’s WDEF resource for the Finder.

6.1 The Williamson Array
6.2 The Goethals-Seidel Array

8.1 Segment of pseudocode illustrating generation of combinations for testing Williamson’s condition.
Chapter 1

Introduction

There are many forms of malicious software; perhaps the best known of these is the computer virus. Computer worms, which in some ways are very similar to computer viruses, may also be familiar.

Computer viruses have existed as a practical threat to computer systems for over ten years. A large number of viruses are known to exist for computers descended from the original IBM PC. A much smaller number of viruses are known to exist for Macintosh computers. Very few viruses are known for UNIX-like operating systems, however it is clear that these systems are far from immune. Computer viruses have also been implemented in the macro language of application packages such as Microsoft Word, and for the most part are able to run successfully on any computer platform to which the host application has been ported. Recently, the trend towards ubiquitous network connection has been exploited by writers of malicious software to create some very rapidly-spreading viruses and worms.

Computer viruses and other forms of malicious software have been extensively studied both from a theoretical standpoint, and from a more practical one with the aim of detecting and eradicating a computer virus infection. Strategies employed both by anti-virus software to detect computer viruses and by computer viruses to evade detection have become very sophisticated.

In Chapter 2, we look at how computer viruses work, and the techniques that are used by anti-virus software to detect viruses. We look in some detail at techniques used by computer viruses on Macintosh computers, and the effects that operating system changes have had on these viruses.

In Chapter 3, we examine an attack that allows the infection of Macintosh application programs without altering existing files in any way. We also suggest a simple
countermeasure to the attack. This attack is similar in spirit to one known for DOS for some time. We consider the utility of being aware of potential security problems to outweigh the risk that the discussion may be used to implement new computer viruses.

This chapter is based on a previously published paper:


95% of the work was performed by Jeffrey Horton and 5% by Jennifer Seberry.

In Chapter 4, we extend the attack described in Chapter 3 and comment on difficulties in the implementation of Macintosh viruses composed of native PowerPC code. We also comment on the future of viruses for this platform in light of changes expected as a result of coming major operating system revisions.

In Chapter 5, a technique for implementing a covert distributed computation using downloadable applets written in Java is examined. Such software would be classed as malicious, as its intent is to steal computational resources from unsuspecting computer users.

This chapter is based upon a previously published paper:


95% of the work was performed by Jeffrey Horton and 5% by Jennifer Seberry.

In Chapters 6, 7 and 8 we alter our focus to examine some problems to which distributed computing, covert or otherwise, could be applied. Using distributed computing to search for encryption keys is a very well-known application. We look instead at some combinatorial search problems.

Chapter 6 provides an extensive mathematical background to the material subsequently covered in Chapters 7 and 8. Some examples of practical uses of sequences and Hadamard matrices are provided.
In Chapter 7 we search for sequences with zero autocorrelation function. A number of necessary conditions are known which these sequences must satisfy; however, these necessary conditions are not by themselves sufficient: the autocorrelation function must be evaluated. We find a number of new sequences, and demonstrate the non-existence of others.

This chapter is based upon a published paper:


80% of the work was performed by Jeffrey Horton and 20% by Jennifer Seberry.

In Chapter 8 a distributed search for Hadamard matrices that uses Williamson’s method is described. Hadamard matrices have extensive applications in many areas. They may, for example, be employed in coding and communications. We have searched for Hadamard matrices in all orders up to and including order 148. These searches consume considerable computer time: the initial search of order 148 required over a month of time on in excess of 20 computers.

Although we do not find any new matrices, these searches are still important, because they provide independent verification of other results. Some previous exhaustive searches for Hadamard matrices have produced incomplete results.

This chapter has formed the basis of a paper which has been submitted for consideration for publication. 90% of the work described was performed by Jeffrey Horton and 10% by Jennifer Seberry. Our results for order 100 have also been independently verified by Christos Koukouvinos, with whom we collaborated when writing the paper.
Part I

Computer Viruses and Other Forms of Malicious Software
Chapter 2

Introduction to Computer Viruses

2.1 Introduction

Computer viruses are not a new problem in the area of computer security; computer viruses have been causing problems for users of personal computers at home and in the business and education communities since the mid 1980s, and have been discovered on computers around the world. Users of personal computers, be they IBM PC compatibles, Macintoshes or something else, are all at risk from computer virus infections.

A commonly accepted informal English definition of a "computer virus" is due to Dr. F. Cohen, as follows:

We define a computer "virus" as a program that can "infect" other programs by modifying them to include a possibly evolved copy of itself. [1]

The key property of a computer virus here is that it infects another program in some way. There is, for example, no requirement in this definition that a computer virus be capable of performing any actions beyond being able to infect them.

Computer viruses are often described as one form of malicious software or malware. Certainly many computer viruses are written with malicious intent. The Brain virus, one of the first viruses affecting the IBM PC, was capable of destroying data describing the location of sectors making up files on a diskette, and might even overwrite parts of a file in the process of infection [2, 3]. Many other types of malicious action are possible, and a great many have been explored by writers of computer viruses over the years.

Furthermore, the nature of what is classified as malicious behaviour varies depending on the environment in which the behaviour occurs. For example, a virus that asks
permission before infecting an executable file might perform what its creator considered was a useful function, but to those whose work is interrupted by the virus, it is a time-wasting nuisance. There is the potential that interruptions to some critical process could even be life-threatening.

Considerable effort has been devoted to the problem of detection and removal of computer viruses. Techniques and strategies used by anti-virus software will be covered in Section 2.6.

While many computer viruses have been created with malicious intent, it should be noted, however, that nothing in the definition above requires that a computer virus be malicious.

Some supposedly benevolent uses of computer viruses have been proposed. Cohen presents as examples a virus that compresses executable files on infection and which decompresses the file upon execution [1], or the potential use of viruses in implementing a distributed database [4]. These ideas and others have failed to win popular acceptance, however.

Why is "possibly evolved" included in the definition? The key property of a computer virus is that it is capable of infecting another program in some way — there is no requirement that the program being infected be infected with a literal copy of the computer virus. In fact, there are a variety of computer viruses that alter the code of copies of itself when infecting some other program. This is a common technique used by computer virus writers to make detection of the virus by anti-virus software more difficult.

One shortcoming of the definition above is that it fails to encompass a program that is able to attach itself to a host program by some means other than altering the code of the host program, but which otherwise would seem well-described by the tag of "computer virus". The "companion" strategy of infection is an excellent example of this. For this reason, the definition above can be extended to include as viruses programs that infect by alternatively modifying the environment in which the host program exists.

The definition above does not require that the result of a virus infecting an executable file is itself a virus capable of further replication. This is an ambiguity that is present in this informal definition — a formal mathematical definition exists [5] which
does not suffer from this ambiguity. The informal definition is adequate for purposes of discussion.

A virus that contains code to perform some additional actions beyond merely replicating is often referred to as having a payload. A virus' payload might be triggered by, for example, the computer system's clock reading a certain time, or on a particular day. There is no requirement that a virus have a payload.

Computer viruses and computer worms are often confused. The distinction between them is, however, sometimes unclear.

Computer worms made their first appearance at the Xerox PARC as a tool for implementing distributed tasks across a network of workstations. The task might be the performance of a distributed computation, such as producing multi-frame animation with the workload spread over multiple machines, or perhaps as a network diagnostic tool [6]. Here worms were intended as a tool to perform useful work.

Perhaps the most famous worm, however, is the Internet Worm that caused much disruption to the Internet in November of 1988 [7, 8, 9, 10]. Despite some arguments [7] that this incident was caused by a program that is better classified as a virus, it has become generally accepted that the program responsible for the incident was best classified as a worm.

The Internet Worm exploited a number of known security holes in some implementations of the UNIX operating system of the time in order to spread itself from machine to machine. Bugs in the worm's code meant that it consumed so many resources on infected computers that it was easily noticed. Less than 5% of the machines on the Internet at the time were affected for a period of several days [10], however.

Other notable worm incidents include the WANK and OILZ worms [11].

As many incidents have occurred with worm programs running out-of-control on computer networks, wasting bandwidth and causing denial-of-services to legitimate users at the very least, it is unsurprising that worms have also come to be referred to as another type of malicious software, despite the ideas behind the original work on worms.

So how might a worm be defined? There exists a formal definition of computer worms along with proofs of some results using this definition [12], but here an informal
English definition is likely to serve best. We define a worm as a self-replicating, self-contained program that is capable of spreading itself to other machines. A network is often involved in the replication process. It may or may not be capable of self-initiation on the remote machine. Unlike a virus, a worm does not “infect” or otherwise depend on a host program — it is self-contained.

A more recent worm incident occurred with the discovery of the AutoStart worm for the Macintosh [13]. This was a self-contained program that exploited the ability, provided by the operating system, to designate a program on a diskette, removable media or hard disk to be executed when the disk was mounted by the operating system. It relied on the transfer of infected media from machine to machine — it did not use the network to spread. It did not depend on a host program in the same sense as a virus. There is, however, some discussion as to its appropriate classification.

The final class of malicious software to be covered here is the Trojan Horse. This is a program that claims to perform a particular function, sufficiently attractive to the computer user to ensure that the user executes the program; instead of or perhaps in addition to performing this function, a trojan horse program takes some form of undocumented action, often malicious, that was intended by the programmer\textsuperscript{1}. Trojan horses, unlike viruses or worms, do not replicate themselves. The action taken by a Trojan horse might, however, be the installation of a virus.

A Trojan horse may be much more subtle than simply deleting files or formatting a disk; they need not be obviously malicious. It is not difficult to imagine a Trojan horse that collects private or personal information (like PGP\textsuperscript{2} private key files, for example) from a user’s hard disk, and sends this information out over the Internet for collection at a remote site. The Trojan horse program would otherwise perform a useful function. A Trojan horse like this would be very difficult to detect.

The name “Trojan horse” comes of course from the events surrounding the fall of the city of Troy — the Greeks, unable to reduce the city by siege, ultimately invaded the city concealed inside a wooden horse, a supposed gift to the Trojans from the Greeks to mark the abandonment of the siege.

\textsuperscript{1}This is distinguished from a simple bug in a program that results in some form of destructive event.

\textsuperscript{2}PGP is an acronym for Pretty Good Privacy, a program which among other abilities is used to facilitate secure and/or authentic exchange of e-mail using public-key cryptography.
2.2 A Little Bit of Theory

It can be shown using a formal definition of a computer virus that it is impossible to distinguish with 100% accuracy between a program that is a computer virus, or is infected with one, and one that is not [5, 4, 14].

What does this mean for writers and users of anti-virus software? As it is impossible to always correctly determine whether or not a given program is a virus, there will be some number of false positives and false negatives occurring.

A false positive occurs when anti-virus software identifies a given file as having been infected with a virus, and is mistaken. That is, the file is not infected with a virus.

A false negative occurs when anti-virus software determines that a given file has not been infected with a virus, and is mistaken. That is, the file has been infected with a virus.

So the goal for developers of anti-virus software is to minimise the number of false positives and false negatives produced by their software package. This can work quite well in practice — theory says that 100% accurate identification is impossible, not that it cannot be very closely approached.

2.3 Platforms Affected

Users of IBM-compatible PCs are the worst affected by the computer virus problem. There are claims [15] of more than 10,000 DOS-based computer viruses having been created as at November 1996! Many of these, however, would be variations based on some original virus strain. The virus problem is certainly worse now — the figure above neglects macro viruses entirely.

Macro viruses are a recent development in the field of computer viruses. Previous to this development, a computer virus was capable only of infecting a particular hardware platform — a computer virus intended to run on an IBM-compatible PC was no threat to a user of a Macintosh, for example. Macro viruses are written in interpreted languages supplied by some common programs that are available across

---

3A Macintosh user running a program designed to closely emulate an IBM-compatible PC would in many cases be at risk from a computer virus designed for that platform — the emulated PC would be infected, but Macintosh executables and files would not be at risk.
multiple platforms. The intention was that these languages should increase the power and flexibility of the program. A good example is Microsoft Word — it is available for both IBM-compatible PCs and Macintosh computers, and macro viruses for this package can affect both PCs and Macintoshes.

Macro viruses have now become one of the commonest types of computer virus.

Macintosh users have also been affected by the computer virus problem, but not to the same extent as users of IBM-compatible PCs. Estimates vary, but there are certainly fewer than 100 viruses targeted specifically at the Macintosh computer at this time.

UNIX users are fortunate in that there are no common virus threats against this platform. However, the potential exists for viruses to be written for this platform — there has been some practical explorations of the possibility by researchers [1, 4, 16, 17]. Some experiments with computer viruses were performed by Cohen as part of his original research on computer viruses, and demonstrated that access to the entire system was achieved by the attacking virus in under one hour [1]. Viruses required a modest number of lines of code to implement, and could be programmed quickly. Similar results were achieved on related platforms.

There exist some threats against users of other computing platforms. However, these are of minor interest only. In principal any computer platform where programs are stored on modifiable media is subject to attack by computer viruses.

2.4 Computer Virus Types and Strategies

A computer virus can only become active on a computer system if the virus is executed. Part of the definition of a computer virus states that a computer virus is attached to a host executable — if the host is executed, then the virus would also be executed. Virus code that has been added to a non-executable object (such as a plain text file) poses no threat.

A few years ago when talking about non-executable objects and computer viruses, we could have said "like a word-processor file". Unfortunately, word-processor files produced by some common word-processing packages can now contain a form of executable code! There are other common packages that have a similar problem. This
development has allowed viruses to be written for the environment provided within each of these packages. Such viruses are called "macro viruses". They are a relative newcomer to the computer virus field, and have proved to be very successful. They are a problem for users not of a particular hardware platform, but for users of the particular application package for which the macro virus is written, regardless of the operating system or hardware on which that package is running.

For any given operating system release and computer platform, there are commonly a variety of different executable objects that might be the target of infection by a computer virus. New operating system releases or even application packages can add new types of executable object.

Many viruses employ a number of strategies that are intended to make the detection of the virus more difficult. We'll consider two of these strategies — polymorphism and stealth.

Computer viruses are a major problem for users of IBM-PC compatible computers, so we'll look at how viruses operate on this platform first. Computer viruses are currently much less of a problem for users of Apple Macintosh computers, so the operation of viruses on this platform will be examined next. Finally, macro viruses will be examined.

### 2.4.1 IBM-PC Viruses

Viruses targeting this platform can be divided into two major groups, with some overlap:

- File infecting viruses;
- Boot sector infecting viruses.

A virus that is capable of employing either strategy — infecting both executable files and boot sectors — is referred to as a multipartite virus. Natas [18] is an example of a multipartite virus, capable of infecting both executable files, such as .COM and .EXE files, and boot sectors.

DOS file infecting viruses and boot sector infecting viruses were each responsible for about 50% of reported virus incidents when running some version of DOS. Following the introduction and widespread acceptance of Windows, however, boot
sector infecting viruses were responsible for a much higher percentage of virus incidents, despite what might seem to be very limited opportunities to spread, while file infecting viruses declined significantly; this is believed to be because Windows 3.x does not work well, possibly not at all, in the presence of a typical DOS file infecting virus, while in many cases boot sector infecting viruses will coexist with Windows 3.x. Changes in the world's computing environment, which includes both computer hardware and operating system software, has an effect on the viruses and classes of viruses that are common [19, 20].

Now it seems that macro viruses, discussed in Section 2.4.3, are responsible for a much higher percentage of reported virus incidents than any other class of virus\(^4\).

A virus may be either memory resident or non-memory resident. A memory resident virus is so named because it has the ability to remain active in the computer's memory even after its original host program has exited. A non-memory resident virus is active only while its host is executing. A non-memory resident virus might also be referred to as a direct action virus, because the virus has opportunities to infect other objects or perform damage only when viral code is called while its host is running.

We'll also look quickly at some more unusual infection strategies:

- Companion viruses;
- "Link" viruses.

Summaries of the techniques we discuss here, as well as some more exotic techniques that we do not discuss in detail, such as viruses that infect source code with a source form of the virus, can be found in [21, 22, 23, 24].

2.4.1.1 File Infecting Viruses

There are a number of executable file formats for users of MS-DOS and Windows. Common varieties of executable file that readers may be familiar with are .COM and .EXE. Rather than examine in detail how viruses interact with files of each type, which can be confusing both because of the variety and complexity of the available file formats, we'll instead look at a more general picture of how viruses on this platform infect files.

\(^4\)Virus prevalence information gathered from Virus Bulletin; macro viruses responsible for in excess of 65% of reported virus incidents in the months January to December, 1999.
2.4. Computer Virus Types and Strategies

The simplest type of file infecting virus is one which overwrites part of the object that is the target of infection, and does not store the code that was overwritten so that it can restored later when the infected file is executed. Figure 2.1 illustrates this strategy.

1. Program before infection by overwriting virus.

```
Program Code
```

2. Program after infection by overwriting virus.

```
Viral Code
Program Code
```

Figure 2.1: An overwriting virus. The start of the host program’s code is replaced with viral code.

We depict the beginning of the file being overwritten by the virus — in this case, the virus would receive control when the host program is executed. The host program itself would likely be so badly damaged that it would be unable to perform its own functions. However, the viral code might also be placed elsewhere in the file, in the hope that the host program would be able to perform most of its original functions.

However, as the infected program has been partially destroyed by the virus, it is most unlikely to function 100% correctly. The best way of repairing programs damaged in this manner is to restore them from backups.

More sophisticated viruses attach to a host program in such a way that the host program can be repaired by the virus and executed successfully. A simple way of infecting an executable file so that any changes made are repairable is to append the virus code to the end of the file, save the first few bytes of code for later restoration, and replace them with a jump to the appended viral code. When the host program is executed, the viral code receives control first, can repair the code of its host and call it. This process is illustrated in Figure 2.2.

It is also possible to prepend the viral code to the host file. This is illustrated in Figure 2.3. The appended host code must be copied over the viral code and executed. The Necropolis virus is an example of a virus that uses a prepending technique [25].

Appending or prepending the viral code to the host file in such a manner that the host file is repairable means that the host file changes in size. This might be noticed by an observant user or a program monitoring the sizes of executable files, and might
2.4. Computer Virus Types and Strategies

1. Program before infection by appending virus.

   Program Code

2. Program after infection by appending virus.

   Jump to start of viral code
   Program Code Viral Code

3. On execution, control passes to viral code. Virus repairs program code.

   Program Code Viral Code

4. Virus executes original program.

   Program Code Viral Code

Figure 2.2: An appending virus. The start of the host program's code is replaced with a jump to appended viral code; the original start of the host is preserved.

1a. Program before infection by prepending virus.

   Program Code

2a. Viral code prepended, program code shifted.

   Viral Code Program Code

1b. Program before infection by prepending virus.

   P. Code #1 P. Code #2

2b. Viral code prepended; only overwritten program code shifted.

   Viral Code P. Code #2 P. Code #1

Figure 2.3: A prepending virus. The host's code is effectively appended to the viral code.
result in detection of the virus. Are there ways in which a file can be infected in such a way that its length doesn't change, yet the file's code can be repaired by the virus at time of execution?

There are several ways that this might be achieved. The first is to find an area of constant data within an executable file that is large enough to contain the virus body, to record the value that was originally stored there and replace the constant data with the virus code. The Lehigh virus [4, p. 42] [3, 26] operated in such a manner. It infected the MS-DOS command interpreter, COMMAND.COM. If a single area is not available, there may be several smaller areas of sufficient size. Before the viral code executes its host program, the original value stored within the constant area or areas can be restored if necessary.

The second is to store the viral code inside unused spaces within an executable file. Several smaller spaces might be used if a single large space is not available. Some of the more sophisticated executable file formats can require the different components of the executable file to be padded to a particular length, even if only a small amount of the available space is required. This is the technique used by the CIH virus [27].

Viruses that employ techniques such as this may be referred to as cavity viruses, as they make use of the small "cavities" inside executable files to store the virus code to avoid changing the length of infected file.

A third possible technique would be for the virus to compress all or part of the file's original contents so that the virus code can be added without changing the length of the original file. The compressed component can be uncompressed at runtime.

2.4.1.2 Boot Sector Infecting Viruses

A boot sector infecting virus works by infecting the small components of code that are used to help load an operating system from a floppy disk or hard disk when the computer is switched on. This process may be referred to as "bootstrapping". We give a very basic overview of the booting process [28, 29, 30, 21].

The loading of an operating system from disk happens in several stages. In the case of a floppy disk, the first logical sector on the disk, referred to as the boot sector or DOS boot sector, consists of a small program that is responsible for starting the next phase of the process of loading the operating system. In the case of a floppy disk
that doesn’t have an operating system, this sector contains a small program responsible for informing the user that this is not a bootable disk, prompting for the insertion of another disk.

Hard disks, because of their large physical size, are often divided into a number of smaller logical chunks referred to as partitions. Reasons for dividing a hard disk into smaller partitions would include a desire to be able to install multiple operating systems, such as MS-DOS and Linux, on a single hard disk, or to break a disk that is too large to be handled by a particular operating system into smaller chunks. In this case, the first physical sector on the disk contains a record of the partitions into which the disk has been divided, and a small program responsible for locating a bootable partition and booting from that partition. The first physical sector of the hard disk is referred to as the Master Boot Record (MBR) or Master Boot Sector (MBS). The first logical sector of a bootable partition is then the boot sector that is used to load the next stage of the operating system.

Figure 2.4 shows a very simple illustration of the boot process for floppy disks and hard disks.

![Diagram of boot process](image)

**Figure 2.4: Simple illustration of boot process for floppy and hard disks.**

The booting process may be customised by modifying the DOS boot sector of a floppy disk, or either of the Master Boot Record or DOS boot sector present on a hard disk. For example, users with multiple operating systems installed on different partitions on their hard disks might install a small program which allows the operating system to be used for this session to be selected during the booting process.

The basic operation of a boot sector virus is then very plain: these viruses infect the code found in the Master Boot Record for hard disks or in the DOS boot sector for floppy and hard disks. The original contents of the infected sector are commonly stored elsewhere so that the virus can continue the boot process using the original boot
sector code. However, it is not unknown for a virus to dispense entirely with the code from the part of the boot sequence it modifies, and to attempt to perform the boot functions itself. The AntiCMOS virus functions in this manner [31].

Note that it is unnecessary for a boot sector virus to be capable of infecting a hard disk — the Brain virus was once very successful even though it only infected floppy disks [2, 3]. However, the inadequacy of the floppy disk for today’s operating system and data storage needs means that a floppy-only boot sector virus is unlikely to be very successful.

Even non-bootable floppy disks have a small program in the boot sector to prompt for the insertion of a bootable disk, so these can be infected by a boot sector virus.

Commonly, PCs will check the floppy drive first, searching for a disk from which to boot. If an infected floppy disk has been left in the drive, then the PC will attempt to boot from the floppy drive, causing the viral code to be executed. Many viruses will infect a hard disk, if one is present, at this point. So it is certainly desirable to avoid booting from a floppy disk if one is unintentionally left in the drive.

It is frequently possible to configure a PC so that an attempt will be made to boot first from the hard disk, rather than checking for the presence of a floppy disk first. Configuring a PC in this manner can help prevent infection by a boot sector infecting virus.

Removal of boot sector infecting viruses can be difficult. For example, a virus could encrypt the partition table data, or copy it elsewhere, so that this information is available only when the virus is present. The Monkey virus [32] is a well-known boot sector virus that copies the partition table data elsewhere, so that an infected hard disk is inaccessible if the computer is not booted from the virus-infected hard disk. The Hare Krsna virus [33] is another example of a virus that tampers with the location of the partition table data.

Finally, some boot sector viruses are able to infect a hard disk by altering not the code in the Master Boot Record, but the partition table data! This technique is used by the Starship virus [22] and, more recently, by the Nutcracker.ABO viruses [34].
2.4.1.3 Companion Viruses

A companion virus is of interest because it does not modify any of the files which it infects. Instead, a separate executable file is created to hold the virus body. Implementations of such a virus depend on the operating system; two basic types of companion virus which could be created under MS-DOS are [35, 36]:

**Regular Companion [35] or Corresponding File Virus [36]:**

Creates a file in the same directory as the target of infection but with a filename extension which the operating system chooses to execute before that of the original file when the extension is not explicitly specified (for example, under MS-DOS a `.COM` file with the same name as a `.EXE` file and in the same directory is executed before the `.EXE` file if the file extension is not specified [29, p. 15]).

**PATH Companion:**

Create a file with any executable extension in a directory that is searched for executable files before the directory containing the target of infection. Named after the PATH environment variables found in operating systems such as MS-DOS and UNIX.

Magruder [36] also discusses "surrogate file viruses", a type of companion virus which renames the executable file being infected and replaces it with a copy of the virus program.

2.4.1.4 "Link" Viruses

This technique is now of mainly academic interest. It represents another way in which a virus can infect an executable file like a `.COM` or `.EXE` file without making any changes to the files themselves.

The MS-DOS operating system records which disk clusters are occupied by a particular file by means of the **File Allocation Table**, or FAT [29, 30]. A cluster may be used by a particular file, in which case it holds a record of the next cluster in the file, or may be unused or bad. A file's directory entry records the file's first cluster.

A link virus, such as DIR-II [37, 23], replaces the starting cluster in an executable file's directory entry with that of the virus, saving the original starting cluster elsewhere
for later use. A request to execute the infected file results in the virus code being run instead, as the starting cluster recorded in the infected file’s directory entry points at the starting cluster of the file containing the virus code. The virus can then arrange for the execution of the infected file.

Many executable files on a disk may be infected with a single copy of the virus code.

### 2.4.2 Apple Macintosh Viruses

#### 2.4.2.1 Introduction

The first Macintosh viruses appeared in the late 1980s. There has, however, been little activity since then on the part of Macintosh virus writers when compared with their IBM PC counterparts. There are many thousands of PC viruses; at the present time, there are fewer than 100 viruses “native” to the Macintosh.

Techniques used by Macintosh virus writers are in some ways less sophisticated than those used by their PC counterparts. For example, the first Macintosh viruses to exhibit even crude polymorphic techniques, members of the *SevenDust* family of viruses, appeared in late 1998. Polymorphic techniques are discussed in more detail in Section 2.4.4.1. These techniques have been popular with IBM PC virus writers for many years. Furthermore, the boot sector virus, a variety of virus that has been very successful on the IBM PC, has no Macintosh counterparts at present.

As a result of the slow creation of new Macintosh viruses, and the limited popularity of the platform in the past, when compared with that of the IBM PC, information on Macintosh viruses can be very hard to find. A very useful, although somewhat dated reference for Macintosh viruses, which goes into some detail not only of the workings of Macintosh viruses, but also potential threats, is [38]. Another source, the Computer Virus Catalog [39], also contains some useful and detailed information on Macintosh viruses; again, it is somewhat out-of-date. Both sources include out-of-date PC virus information. Some more current but still out-of-date information on Macintosh viruses can be found in [40].

The variety of techniques used by Macintosh viruses in the process of infection make viruses on the Macintosh of considerable interest. Changes to the Macintosh operating system have rendered techniques used by some once effective and widespread
viruses, like the WDEF virus, ineffective. Some Macintosh viruses, even those written many years ago, are still capable of posing a threat to modern versions of the operating system. Additionally, new opportunities for virus infection have become available; these will not be examined in detail here, however.

Before making an attempt to cover how Macintosh viruses operate, some background on their operating environment, the Macintosh operating system, is required. There are very distinct differences between the Macintosh and PC operating systems from a programming perspective, despite their similarities.

When discussing details of the Macintosh and viruses affecting the Macintosh, a typewriter font like this will be used when talking about resource types, and a bold font like this will be used when talking about viruses, some of which happen to have been named after a resource type used by the virus writer to implement an attack, to clearly distinguish between the two.

2.4.2.2 Macintosh Files

Every Macintosh file is composed of two components. These are referred to as forks. Each file has a data fork and a resource fork, also referred to as a resource file. [41, p. 1-4]

The data fork is used to store a file’s data. The format of the data stored in a file’s data fork is at the discretion of the application program that created the file — to the operating system, it is merely a sequence of bytes. [41, p. 1-5]

The resource fork is used to store a file’s resources. System software expects that a file’s resource fork will have a certain internal structure. There are a variety of system software routines that manipulate the contents of the resource fork. Resources are discussed further in Section 2.4.2.3.

In the case of an ASCII text file, for example, the data fork would be used to store the file’s text. The resource fork could be used to hold resources that described the cursor position and window size when last the file was edited, and formatting information like the font and font size used by the text of the document.

Another oddity of the Macintosh file system is that with each file is associated a type and creator. These are both expressed as four-letter codes.

Common examples of file types are:
APPL  This type indicates that the file is an application program. There are other similar types; this is the commonest.

TEXT  Files of this type usually consist of plain ASCII text.

A file's creator indicates the application program that "owns" the file — this is the application program that will be executed when the file is opened by double-clicking on the file, for example\(^5\). Application programs are expected to have unique creator codes; this ensures that the correct application will be used to process a file marked with that creator. Many different file types may be associated with a single application program.

2.4.2.3 Resources

A resource can contain data of any description. Resources within a resource fork are described by:

- A resource type, which is a four-letter code;
- An ID number, which is a two-byte integer;
- A name, which is a string of characters (optional).

To identify a particular resource, it is sufficient to specify a resource type and either an ID number or a resource name.

What sort of data is commonly stored in a resource? A variety of information is stored in an application's resource fork, for example:

\textbf{MENU}  Stores information about the items in a particular application menu. For example, a \texttt{MENU} resource for an application's "File" menu would contain items such as "Open" and "Save".

\textbf{MBAR}  Lists the menus that are present in an application's menu bar.

\textbf{WIND}  Describes the dimensions and other characteristics of a window created by an application.

\(^5\)Files may also be opened in other ways.
CNTL Defines a control, which is a user interface element such as a button, checkbox or scrollbar, created by an application.

CODE Contains main components of an application’s executable code.

There are many other standard resource types. Developers are also free to create their own resource types to hold data specific to their application.

Programmers on other computer platforms will be familiar with the way in which operations, such as reading data from a file, are performed on a file — the programmer specifies explicitly the file on which the operation is to be performed. This is also the case when working with the data fork of a Macintosh file.

Some operations on resource forks are a little different, however. When loading a resource from a file into memory, for example, more than one file can potentially be involved — a search path is followed to locate the resource. The first file to be searched is referred to as the current resource file. The current resource file is usually the resource file most recently opened by an application. An application’s resource file is opened by the operating system when the application is executed, so it is present in the resource search path. The final entry in the resource search path is the so-called “System” file. The System file contains resources that form a part of the operating system and are used by many application programs. [42, p. 1-10]

So if a resource is not found in the current resource file, resource files earlier in the search path will be searched for the required resource. The search stops when a resource is found, or all the files in the resource search path have been examined without result. [42, p. 1-10]

This search scheme enables applications to override resources that are present in the System file with application-specific resources, or, if the application has been designed to accommodate it, to override resources in the application with document-specific resources. Remember that the search for a resource stops as soon as one is found. [42, p. 1-10]

Resources that contain executable code are of particular interest in any discussion of computer viruses on the Macintosh. There are a large variety of such resources, only some of which will be examined here. CODE resources, which contain much of an application’s executable code, have already been mentioned above. First resources containing executable code used to draw graphical user interface elements will be
The graphical user interface presented by an application can be customised to a degree by an application developer. How is this done? Many of the graphical user interface elements, such as menus, windows and controls like buttons, checkboxes and scrollbars, are drawn by definition procedures. The executable code of a definition procedure is stored in a resource and loaded by the system software when required to draw a user interface element. The system software provides default implementations of the definition procedures — an application only needs to provide an implementation of a specific definition procedure if different functionality is required.

Some of the definition procedures that are involved in presenting the graphical user interface are:

- **A menu definition procedure** is stored in an MDEF resource, and is responsible for the drawing of menu items within a menu [43, p. 3-9]. The menu definition procedure that is to be used to draw a particular menu is specified by resource ID in the menu’s MENU resource [43, p. 3-88].

- **A menu bar definition procedure** is stored in an MBDF resource, and is responsible for the drawing activities related to the display of menus, such as saving or restoring the portions of the display behind a menu [43, p. 3-9] [43, p. 3-87].

- **A window definition function** is stored in a WDEF resource, and is responsible for such tasks as drawing the window frame or resizing a window. The WDEF resource to be used by a particular window is specified by resource ID in the window’s WIND resource. [43, pp. 4-120-4-127]

- **A control definition function** is stored in a CDEF resource, and is responsible for such tasks as drawing the control and testing for where the mouse has been clicked by the user within a control. The CDEF resource to be used by a particular control is specified by resource ID in the control’s CNTL resource. [43, pp. 5-109–5-115]

Another important type of resource containing executable code is the INIT resource. These are resources that contain code that is intended to be executed at system startup time, to provide extra services or modify existing operating system services via
2.4. Computer Virus Types and Strategies

a mechanism that will be discussed in Section 2.4.2.6. INIT resources can be located within the System file itself, or in files of particular types, for example of type INIT$^6$, within the System folder, Extensions folder or Control Panels folder. A now somewhat dated description of the system startup procedure, which includes more information on INITs and the process of INIT loading, can be found in [44, Ch. 9].

2.4.2.4 The Macintosh Finder and the “Desktop”

The Finder is an application that is provided with the Macintosh system software and which works with the system software to manage the display of the computer user’s “desktop”, keep track of the location, both on the screen and on the disk’s directory structure, of files and folders as they are manipulated by the computer user [42, p. 7-3] [45, p. 6]. The Finder also ensures that the appropriate application is used to work with files which it has created when the file is double-clicked by the user, for example. Figure 2.5 shows a Finder desktop display with some open windows.

![Finder desktop display](image)

Figure 2.5: Finder “desktop” display with some open windows.

Under Macintosh system software prior to System 7, “Finder” refers to a version of the software that would permit only one application at a time to execute. That is, users could run the Finder or some other application, but not both at once. “MultiFinder”

---

$^6$ INIT resources may also be located in files of certain types other than INIT. However, this will not be examined in detail here.
2.4. Computer Virus Types and Strategies

refers to a refinement of the Finder that would permit more than one application, including MultiFinder itself, to execute at a time. It otherwise performs the same task as does Finder. MultiFinder was introduced with System 5 and was subsequently available with System 6 [46]. System 7 and later use "Finder" to refer to a version of the software descended from MultiFinder. System 7's Finder allows more than one application at a time, including the Finder, to be executed. All subsequent releases of Finder had this ability.

Application developers are able to designate an icon for each type of file that is created or "owned" by that application; these icons will be displayed by the Finder to represent the user's documents. Icon information is provided by resources from the application's resource fork; the Finder extracts this and a variety of other information from the application's resource fork, and stores it in a database for easy access. The location of the application on disk is also stored.

To determine the icon to display for a document, the Finder checks its database for an icon corresponding to the document's type provided by the application with the same creator code as the document. When a document is double-clicked by the user, the Finder searches its database for an application whose creator code is the same as the document's creator code, and if found executes that application to process the document.

Under system software prior to System 7, the Finder's database was stored in the resource fork of a file named "Desktop". This was an invisible file located in a disk's root directory. System 7 and later store the Finder's database in the data fork of two separate files, also invisible and located in a disk's root directory, unless the disk's size is less than 2 megabytes, in which case the old pre-System 7 "Desktop" file is used [42, pp. 9-3–9-4]. The Finder's database is more properly referred to as the "Desktop Database".

Several viruses have been created that infected the Finder's desktop database. We discuss how these viruses worked in Section 2.4.2.7.

The Windows "registry" is used to perform a similar function to the Finder's Desktop Database. A document file's creating application is identified only on the basis of the document's filename suffix, however. This means that only one application program can be associated with each file suffix. The strategy used by the Macintosh allows
many applications to create the same types of files, if appropriate\(^7\), but for each file to be associated with the appropriate application. The Desktop Database may also be rebuilt if damaged simply by causing the Finder to scan a volume for application files when mounting the volume.

### 2.4.2.5 Macs and Microprocessors

The Macintosh was first introduced in 1984. It was based on the 68000 microprocessor from Motorola. Many later Macintosh models were based upon the 68020, 68030, 68040 and 68LC040 microprocessors. Macintoshes based upon the 680x0 series of microprocessors are often collectively referred to as "68K Macintoshes", and code intended to run on these microprocessors as "68K code".

In 1994, the first Macintosh computers based on the PowerPC series of microprocessors were introduced. Ultimately, PowerPC microprocessors replaced 680x0 microprocessors in new Macintosh models. Macintoshes based upon a PowerPC microprocessor are often collectively referred to as "PPC Macintoshes", and code intended to run on these microprocessors as "PPC code".

The machine code of the 680x0 microprocessors was completely incompatible with the newer PowerPC microprocessors. This would present something of a problem in the transition from 68K to PPC, as old applications would be useless on a PPC Macintosh and would need to be replaced immediately. To avoid this, and allow computer users to change to PPC-native applications at their leisure, Apple provided a 68LC040 emulator as part of the system software of the PPC Macintosh machines [47, Ch. 1] that would allow 68K code to execute on a PPC-based machine. The emulator also meant that the implementation of operating system components where performance was not critical could be deferred to a later time.

The performance of a program executing under emulation on a PPC Macintosh is obviously inferior to the performance of native PPC code running on the machine. It was often advantageous to build an application that contained 68K code and PPC code, and was therefore able to be executed on Macintoshes based on 68K or PPC microprocessors. The two code components are stored separately within the executable file.

\(^7\)TEXT files, containing ASCII text, are common.
2.4. Computer Virus Types and Strategies

The manner in which the executable code of an application program is stored within the application file is extensively documented in [48]. There are three basic code models:

- application consisting of 68K code split into a number of code segments; the "classic" 68K runtime architecture. This variety of application can run under emulation on a PPC Macintosh.

- application consisting of PPC code, based on code fragments.

- application consisting of 68K code, based on code fragments, but typically also containing a small code segment based component, responsible for starting up the code fragment component on versions of the operating system that are unable to automatically perform this task. This variety of application cannot run under emulation on a PPC Macintosh.

Either of the two 68K code models may be combined in the one application with the PPC code model. This results in an application which is able to run on a 68K or PPC based Macintosh. On a PPC Macintosh the 68K code is ignored in favour of the PPC code. A 68K based Macintosh of course ignores any PPC code that may be present.

Viruses seen so far on the Macintosh that infect by modifying an application’s code only infect the 68K code segments. Why then do users of PPC Macintoshes need to be concerned with viruses?

- Many PPC applications contain at least a small 68K code segment based component which indicates to the user that the application cannot be run on a 68K Macintosh; some will contain 68K code for the entire application. This 68K code segment based component can be infected with a virus and poses a danger to a user of a 68K Macintosh who should happen to run the application. This 68K component may also be executed on a PPC Macintosh if the PPC code is intentionally removed.

- Not all viruses work by modifying the application’s code directly. Some add executable resources like the MDEF resource to an application to infect it with the virus. 68K definition procedures added to a PPC application by a virus will still work in a PPC environment.
• While there are currently no viruses that infect an application by modifying PPC code fragments, there is every reason to suppose that such viruses will be written in the future.

As there are currently no viruses that modify code fragments, this code model will only be examined briefly. Code fragments are typically stored in the data fork of an application file, but may also be stored in resources in the resource fork. A resource known as the code fragment resource, or cfrg resource, with ID 0 is used to index the fragments stored in the data fork or resource fork.

Applications based on code fragments can also import shared libraries. These might be more familiarly known to IBM PC users as dynamically linked libraries, or DLLs. The code fragments within a shared library could potentially be infected by a virus; no such viruses for the Macintosh are currently known.

A code fragment based runtime environment is a more flexible and programmer-friendly environment than the 68K code segment based runtime environment. With these advantages, however, comes greater complexity, which may have deterred Macintosh virus writers to date.

Figure 2.6: Simple illustration of a code segment based 68K application. The jump table (CODE 0) contains references to the entry points of routines that can be the subject of an intersegment reference.

The structure of a code segment based 68K application is much simpler, however. Figure 2.6 is a simple illustration of a code segment based 68K application. The executable code of a 68K application based on this model is usually divided into a number of segments, each segment stored in a CODE resource. This has the advantage
that programs can be constructed so that only a subset of these resources is required in memory at a particular time; CODE resources not currently required can be removed from memory. Large, complex programs could then be executed in limited memory.

A mechanism is needed to enable a routine located in one code segment to call a routine in another code segment. The code segment containing the routine being called may need to be loaded into memory. Intersegment references are handled by means of a structure called the **jump table**. To call a routine in another segment, a jump is executed to the jump table entry corresponding to the routine to be called. If the routine’s code segment is not already present in memory, this entry contains a small snippet of code to load the segment into memory and then jump to it. Otherwise, the jump table entry contains a jump to the location in memory of the routine. If a segment is unloaded from memory, the jump table entries of routines in that segment are restored to their unloaded format.

The jump table is stored in the CODE resource of ID 0. The first entry in the jump table indicates the routine that receives control when an application first begins executing. Not all compilers store the entire jump table in the CODE 0 resource; instead, an appropriate jump table is constructed in memory on execution, using data stored by the compiler. There are also different formats possible for the jump table, to support extra application capabilities. However, the first entry, specifying the routine that receives control when the application begins executing must always be present in the jump table.

### 2.4.2.6 Organisation of Macintosh System Software

Programs written for the PC invoke system software routines by executing a software interrupt. A table of addresses is maintained that stores the routine to be executed when a particular interrupt occurs. This means that the internals of the operating system can freely be modified by the operating system developers without affecting the function of application programs, which invoke system software routines via a software interrupt.

A similar scheme is used by the Macintosh. For full details on how this is implemented on a 68K Macintosh, consult [44, Ch. 8]. To invoke a system software routine, a program executes an instruction not implemented by a 680x0 microprocessor. This causes an exception handler provided by system software to be executed, which in turn
invokes the appropriate system software routine.

The system software routine that handles the microprocessor’s exception is referred to as the \textbf{trap dispatcher}. An exception resulting from executing one of the unimplemented instructions reserved for the purposes of invoking a system software routine is referred to as a \textbf{trap}; this term may also be applied to the unimplemented instruction itself. The trap dispatcher consults \textbf{trap dispatch tables} to determine the address of the routine to call to perform the operation required by the executing application program.

Many system software routines have their own trap. Some, however, share a single trap. The specific routine that is required in this case is determined using a value from the stack or a register.

The contents of the trap dispatch tables can be modified so that routines other than the normal system software routines are executed in response to a trap. This allows system software routines to be enhanced or replaced, and is referred to \textbf{patching a trap}. The “patch” is the routine replacing the system software routine in the trap dispatch tables. It is not uncommon for a trap patch to call the routine whose address it replaces in the trap dispatch tables.

When using MultiFinder with System 6 or Finder with System 7 and later, each application is provided with its own copy of the trap dispatch tables. This ensures that patches installed by one application don’t interfere with other applications running at the same time, and are disposed of when the application exits. A patch that applies to all applications must be installed during system startup. \texttt{INIT} resources, described in Section 2.4.2.3, commonly contain code to install trap patches at system startup.

Trap patching was a little different under System 6 and earlier when using Finder, rather than MultiFinder. System 6 using Finder was able to run only a single application at a time, either Finder itself or one of the user’s applications. Most significantly, trap patches installed by an application would still be present following the termination of the installing application, unless removed by the application before exiting\footnote{The author was unable to find any explicit written documentation for the behaviour of trap patches in system software before System 7. That trap patches are still present if not removed by an application before exiting was verified experimentally using a Macintosh LC, System 6.0.8 with Finder, and MacsBug 6.5.3, a low-level debugger from Apple Computer. It was also verified that System 6.0.8 with MultiFinder behaves as per documentation describing trap patching under System 7 and later.}. These trap patches would then affect any applications that were subsequently run.
2.4.2.7 How Macintosh Viruses Work

Now that the necessary background information on the Macintosh has been covered, we can look in detail at the workings of Macintosh viruses.

Most Macintosh viruses modify the executable code of an application program in some way so that when the program is run, at some point during execution of the program the viral code is executed. This usually occurs as the first step in program execution. There are two common methods of achieving this goal:

- A virus can alter the code segments of the application by adding a new code segment or modifying an existing one;

- A virus can add resources containing definition procedures that will be invoked implicitly during application execution.

A less successful strategy for a virus is to infect only documents of type INIT. These contain code which is intended to be loaded at system startup time. An example of a virus that employs this strategy is the INIT 1984 virus [39, 49].

Some viruses install patches so that viral code can receive control when particular operating system functions are invoked; the intention is to identify applications which are candidates for infection, and to infect suitable applications. These patches must be installed at system startup when using MultiFinder with System 6 or when using System 7 and later. This task is commonly performed using an INIT resource. Although it is possible to modify an existing INIT resource, the common strategy is to create a new INIT resource containing viral code. This resource may be added directly to the System file, or to documents of type INIT located within the Extensions folder. A new document of type INIT located in the Extensions folder may also be created to hold the resource. There are other places the resource may be located so that it is loaded at system startup, however.

Many Macintosh viruses use INIT resources to install patches at system startup time.

In some places, such as [38] and [39], Macintosh viruses using this strategy may be referred to as link viruses. It is important to distinguish between these viruses and PC link viruses, which are briefly described in Section 2.4.1.4 — unfortunately the names are the same, but the strategies are completely different.
2.4. Computer Virus Types and Strategies

Figure 2.7: Illustration of modifications made by virus that adds an additional CODE resource to an application to achieve infection.

Figure 2.6 is an illustration of a normal 68K application, before it has been infected with a virus. There are a number of ways in which viruses have modified an application’s code segments to achieve infection:

Strategy #1

The virus adds a code segment to the application, in the form of an additional CODE resource, and modifies the first entry of the jump table to refer to the added viral CODE resource, so that the viral code receives control when the application starts executing. The original first jump table entry is saved by the virus so that it can return control to the application once it has completed its work. Other supplementary resources may also be added. The application’s existing code segments, aside from the jump table, are not modified. This is the strategy that was used by the nVIR [50, 51] series of viruses, one of the most successful families of Macintosh viruses [52]. This strategy has also been used by the INIT 29 [53] viruses, the Scores virus [54] and by the CODE 32767 virus. This strategy is illustrated in Figure 2.7.

Strategy #2

Rather than adding an additional code segment to the application, or modifying the jump table, a virus can add its code to the end of an existing CODE resource,

---

10There are a number of very minor variants of the nVIR virus. The nVIR virus was named after the extra resources, of type nVIR, that it adds to an application and that hold other parts of the virus’ code and data. nVIR variants function almost identically to nVIR itself, but add resources with different names. They are therefore named differently.
and modify the first few bytes of a routine in that CODE resource so that when the routine is called, the added viral code is invoked. The bytes replaced by the virus can be saved and restored after running the viral code. This strategy is used by the ANTI family of viruses\textsuperscript{11}. A slight modification of this strategy was used by the T4 D virus: this virus replaces a call to InitDialogs, an operating system function usually invoked by an application when it first begins to execute, with a branch to appended viral code. Other members of the T4 family are similar [39]. The strategy is illustrated in Figure 2.8.

**Strategy #3**

The third strategy that is seen is for the viral code to be added to the end of an existing CODE resource, and the jump table to be modified to pass control to the added viral code. The original jump table entry is saved so that control can be passed to the application after executing the viral code. This strategy is used by the ZUC family of viruses, and is illustrated in Figure 2.9.

There are several ways in which an application can be infected by the addition of a resource containing the code for a definition procedure. Definition procedures are discussed in Section 2.4.2.3. Figure 2.10 illustrates the typical relationship of an

\textsuperscript{11}Note that the ANTI family of viruses were rendered mostly ineffective by changes in the way trap patching was handled when MultiFinder was introduced under System 6. ANTI installs patches containing viral code that seem to be expected to apply to all applications; however, since the introduction of MultiFinder, such patches must be installed at system startup time. ANTI's patches are removed once the installing application exits, so it has very few opportunities to infect further applications. See Section 2.4.2.6 for more details on the subject of trap patching. This is not a comment on the effectiveness of the strategy used by ANTI to infect applications, merely on some technical details of that particular family of viruses that can affect its further replication under operating system versions more recent than System 6 using Finder.
application menu with a default definition procedure\(^\text{12}\). The first way in which such resources can be used in the implementation of a virus is by the addition of a definition procedure with the same type and ID as a standard definition procedure resource from the System file. The resource present in the application overrides that in the System file, also discussed in Section 2.4.2.3. For example, when an application’s menus are created, any MDEFs required for these menus are loaded. Most menus use the System MDEF 0; the presence in the application of a MDEF 0 containing viral code means that it will be loaded and executed in place of the System MDEF 0. This strategy is used by the MDEF and MBDF families of viruses, and is illustrated in Figure 2.11.

The second strategy that can be used is to change the ID of the resource containing a definition procedure to be used for drawing something like a menu. In the case of

\(^{12}\)This relationship can be more complicated. As an example, consider the Appearance Manager, a system software component that was introduced with MacOS 8; versions that were compatible with some earlier versions of system software were subsequently introduced. The System file contains old-style definition procedures, MDEFs and the like, while a separate Appearance component includes updated definition procedures to enhance the Macintosh GUI. Appearance definition procedures typically override the old-style versions contained in the System file; this may affect the correct operation of definition procedure-based viruses that attempt to infect the System file so that the viral code will be executed even when an uninfected application is executed, and may then infect the application. Other third-party system extensions to achieve similar results to the Appearance Manager may also affect the correct operation of such viruses.
2.4. Computer Virus Types and Strategies

Figure 2.11: Illustration of the relationship between an application MENU, a viral MDEF and the System MDEF.

![Diagram](image1)

Figure 2.12: Illustration of the relationship between an application MENU, a viral MDEF and the System MDEF.

![Diagram](image2)

a menu, this information is stored in the menu’s MENU resource. By changing this information, when a menu is created, the viral MDEF resource is loaded and executed. This strategy also appears in the MDEF family of viruses, and is also used by members of the SevenDust family of viruses. Figure 2.12 illustrates the strategy.

By changing the ID of the overridden resource containing the definition procedure in the System file to another value, and adding the resource containing the viral code with the original ID of the overridden resource, the viral code will be executed even when uninfected applications are executed. These applications can then be infected. Figure 2.13 illustrates the relationship of an uninfected application to an infected System file.

![Diagram](image3)

Figure 2.13: Illustration of the relationship between an uninfected application and a System infected with a virus using an MDEF resource.
A combination of strategies is certainly possible. Members of the **SevenDust** family of viruses infect applications by adding a MDEF resource containing the viral code to an application and altering a MENU resource to refer to the new resource. When an infected application is executed, an INIT resource containing viral code will be added to a new or existing file so that the viral code will be executed at system startup. A trap patch is then installed so that applications can be identified for infection.

Special mention is required of the **WDEF** virus [55], once a common and successful Macintosh virus [52]. As the name suggests, this virus infects using a definition resource. However, the definition resource is not added to an executable file!

The **WDEF** virus adds a WDEF 0 resource containing the viral code to the resource file on each disk in which the System 6 Finder stores its desktop database, the "Desktop" file. The Desktop file is discussed in Section 2.4.2.4. This resource file is opened by the Finder when a disk is mounted, which occurs when a floppy disk is inserted, for example, and is held open while the disk is mounted. When the user subsequently opens a window within the Finder, the operating system will search for a WDEF 0 resource to perform window drawing operations; as the most recently opened resource files are searched first for resources, the viral WDEF 0 resource stored within the Desktop file will be found and executed in place of the operating system’s WDEF 0 resource. This is illustrated in Figure 2.14.

![Diagram of WDEF virus strategy](image)

**Figure 2.14:** Illustration of the strategy employed by the **WDEF** virus: a WDEF resource located in a floppy disk’s Desktop file overrides the System’s WDEF resource for the Finder.

---

13 The **CDEF** virus uses a similar strategy to the **WDEF** virus.
The exact circumstances under which the WDEF 0 resource in a disk’s Desktop file is executed is a little more complicated. To perform some simple tests, a custom WDEF resource was written that when executed invoked a low-level debugger to display an indication that the custom WDEF code had been called. On return from the debugger the original system WDEF resource is loaded and called. This custom WDEF resource was inserted into a floppy disk’s Desktop file as WDEF 0. This mimics the non-viral behaviour of the WDEF virus.

It seems that the custom WDEF resource will be executed when opening the infected floppy disk itself or any directory on the infected disk, or Finder/MultiFinder windows not directly related to disks and directories, such as the “About the Finder” window. Opening directories located on disks mounted before an infected floppy was mounted does not appear to result in the execution of the custom WDEF located in the floppy disk’s Desktop file; this is perhaps because the Finder/MultiFinder has altered the resource search order in preparation to displaying the contents of the directory being opened, temporarily removing the floppy disk’s Desktop file from the search path. As windows that are open when a disk is unmounted are re-opened when later the disk is mounted, the viral code could be executed as the result of inserting a floppy disk, without even executing any applications on the disk.

Changes to the Finder in System 7 and later versions eliminated the threat from viruses such as WDEF that added resources containing viral code to a disk’s Desktop file [56]. This is in part a result of shifting from storing the Desktop Database in a resource fork to storing it in the data fork across two files, for volumes over 2 megabytes in size, which meant that hard disks, which tend to have capacities considerably greater than 2 megabytes, were no longer vulnerable to infection. System 6 and the Macintosh models capable of running it are, at the time of writing, very old; many models of Macintosh have since been produced that are not even capable of running System 6. And more recent versions of the operating system do not seem to be vulnerable to the disk Desktop-infection technique used by this virus. So it is to be hoped that this information is of historical interest only.
2.4.2.8 HyperCard Viruses

HyperCard is an application program which allows the creation, modification and "execution" of documents referred to as "stacks". Stacks are composed of one or more "cards", and may use scripts written in HyperCard's scripting language HyperTalk [57] to handle user interaction with a stack or provide other functionality. For example, a script can be attached to a button on a card which will perform actions appropriate to the button when the button is clicked by the stack user.

Executing a script or performing actions such as clicking on a button results in a "message" being sent. Handlers for messages are implemented using scripts. Handlers may be located in places other than the current stack. If a handler for a message is not available in a script attached to the object that is the target of the message, the message will be sent to the next object in the message-passing path. The message-passing path includes the "Home" stack. The "Home" stack is a special stack that is in the message-passing path of all stacks. Handlers may also be present in the "Home" stack, and will be invoked if no more specific handler for a message is available. This fact may be exploited by viruses written in HyperTalk. Some similarities to the way many macro viruses that are hosted by applications such as Microsoft Word are apparent here [58] — a macro virus may copy its code to a location where it is available globally to all documents, so that uninfected documents may be infected with a copy of the virus' code.

Viruses written in the scripting language HyperTalk are referred to as "HyperCard viruses". For more information on the workings of these viruses, consult [58] or [59].

HyperTalk is a flexible, powerful language. However, a script written in HyperTalk does not have the same access to facilities provided by the operating system as does a program written in a lower-level language such as C or Pascal. HyperTalk is extensible by means of resources of type XCMD or XFCN that contain executable code compiled from programs written in a language like C or Pascal. XCMD and XFCN resources are more examples of resources that contain executable code and which could potentially be infected by a Macintosh virus.
2.4.2.9 The AutoStart Worm

In 1998, some malicious Macintosh software was discovered that used a novel technique for infection and transmission of the software from one computer to another. It was classified as a worm, because the software is capable of self-replication, but unlike a virus does not directly attach itself to another executable file in some way.

There are a number of members of the AutoStart worm family. They all use similar techniques. The novel technique used for infection by the worm will be examined here; for a more detailed treatment of the functioning of the AutoStart worm, consult [13].

The AutoStart worm takes its name from the novel technique it uses for infection. The AutoStart [60, pp. 533–534] [61, pp. 625–626] feature is supplied by Apple Computer's QuickTime software. QuickTime is software that can be used by developers to create application programs that work with sounds, still images and movies, and supports a variety of formats for the data. The AutoStart feature, which was introduced with QuickTime 2.0, allows the designation of an application program or document to be loaded automatically when a storage medium such as a CD-ROM is mounted.

This feature was intended to allow an application program or document to be loaded automatically when a CD-ROM is inserted, which may be useful for CD-ROM titles targeted at the educational or entertainment market. However, the AutoStart feature works for other types of media as well, such as floppy disks.

It is very easy to set an application program to be executed when a disk is mounted: it is necessary only to specify the name of the program in the appropriate location in a disk's boot blocks. We refer to the program so designated as the “AutoStart program”. It must be located in a disk's root directory, and it may be an invisible file. It is therefore not obvious that a disk is infected.

The worm spreads from computer to computer via disks where the AutoStart program has been set to be the worm's program file. When the disk is inserted, the AutoStart program is executed and the worm infects the system so that the worm program will be executed whenever the system is rebooted.

Fortunately, it is very easy to prevent infections by this worm. This requires only that the AutoStart feature be disabled. However, this is possible only in QuickTime 2.5 and later.

Although QuickTime has been made available for users of Microsoft Windows,
QuickTime's AutoStart feature is not available to Windows users as it is dependent on certain features of Apple Computer's disk formats.

2.4.3 Macro Viruses

A macro can be thought of as a collection of statements in some language that perform a task when interpreted or otherwise executed. Macro facilities are provided by many application packages to allow users of the package to automate tasks encountered when using the package that are common or repetitive, or to otherwise customise an application package to some degree.

Some application packages provide more than a simple scripting language that can be used to control the application in some basic ways, instead providing an interpreter for a complete programming language. Microsoft Word is perhaps the best-known of these applications. Microsoft Word 6 provided as its macro language a version of BASIC\textsuperscript{14} that was called WordBasic. Microsoft Word 6 was the first implementation of Word to provide this macro language in both the Macintosh and PC implementations.

Perhaps unsurprisingly, virus writers learned how to use the macro language provided by Microsoft Word to create viruses. Writing WordBasic macro viruses is much more straightforward than implementing other varieties of virus, such as a file or boot sector infecting virus. There are a variety of ways in which a Word macro virus may arrange for its execution at some point in time, all of which have been used in a number of macro viruses [62]; some of these are:

- There are a number of macros referred to as "auto macros", that if present in a document or the Word environment are executed when certain actions are performed within Word. These include \texttt{AutoOpen}, which will be executed whenever a document is opened, and \texttt{AutoClose}, which is executed when a document is closed. There are several other "auto macros".

- Macros with names like \texttt{FileSave} or \texttt{FileSaveAs} that are contained within a document will override the corresponding "system" macro. So, for example, a \texttt{FileSaveAs} macro in the active document will be executed when the "Save

\textsuperscript{14}BASIC is an acronym for "Beginner's All-purpose Symbolic Instruction Code". It is a general-purpose programming language once commonly used by novice programmers, because it is relatively simple and easy to learn.
As option is selected from the “File” menu, rather than the normal “system” macro that performs this function.

- Menu items can be replaced with substitutes that invoke a virus’ macro when that menu item is selected by the user.

- Almost any keypress can be associated with a macro. The macro will be executed when its corresponding keypress is detected.

The concept of a macro virus is not new. The possibility of a macro virus was described by Highland in 1989 [63, 64], but it was not until 1995 that one of the first of these viruses, **Concept** [65], appeared and spread widely around the world. There are many useful references describing macro viruses in general and dealing with aspects of the macro virus problem, including [62, 66, 67, 68, 69, 70, 71]. For example, one interesting aspect of some macro viruses is that they are able to incorporate macros from other macro viruses or non-viral macro packages, to create what may be thought of as a “new” virus, or are capable of replicating with a reduced set of macros. Problems such as these can create difficulties for anti-virus developers when implementing strategies to deal with macro viruses.

By the end of March 1999, almost 600 macro virus strains were known, representing almost 3700 macro virus variants in total [72]. By the end of September 1999, 775 macro virus strains were known, which represented almost 4600 total macro virus variants [73]. Most of these macro viruses are targeted at versions of Microsoft Word. A variety of other classes of malicious macro software are also listed, including some macro virus generators, which are small programs intended to be used to automatically generate new macro viruses, and some macro trojan horses.

These viruses are remarkable because they exist as executable code embedded within what appears to the computer user as a document file. Since the macros are interpreted by the application package rather than compiled into machine-specific executable code, they are executed within their host application on any computer hardware platform to which the host application package has been ported. Many Microsoft Word macro viruses work well within Microsoft Word running on a PC or a Macintosh, for example. Differences can arise, however, if a virus attempts to make use of functionality that is provided by its host application on one particular platform and not a similar
version on another platform. This can be the case when the functionality in question relates to access to the underlying operating system, for example.

Macro viruses are now more successful than file or boot sector infecting viruses, responsible for in excess of 65% of reported virus incidents in the months January to December 1999; in nine months of 1999 macro viruses were responsible for in excess of 80% of reported virus incidents\(^\text{15}\). It is likely that these viruses are more successful than file or boot sector infecting viruses because the exchange of documents is arguably much more common than the exchange of executable files, which may be infected with a file infecting virus, or disks that might be infected with a boot sector virus\(^\text{16}\). E-mail enables rapid distribution of infected documents between computers locally and at remote sites.

Later versions of Microsoft Word beginning with the version included with Office 97 for PCs, and Office 98 for Macintoshes, replaced WordBasic with Visual Basic for Applications (VBA). Other applications, such as Microsoft Excel and PowerPoint also provide VBA as their macro language; some versions of Excel released prior to Office 97 supported a version of Visual Basic for Applications.

It is certainly possible to implement macro viruses using Visual Basic for Applications. Furthermore, in the interests of maximum compatibility, the ability to convert macros from Word 6's WordBasic to Visual Basic for Applications was provided. This meant that viruses written in WordBasic could in some cases be converted to viruses that would function effectively in the environment provided by Word 97. Some protection against "upconverting" common WordBasic macro viruses was provided by Microsoft, but was found to be not wholly effective in stopping the translation of viruses from WordBasic to VBA.

As the applications Microsoft Word, PowerPoint and Excel provided with Office 97 share a common macro language, Visual Basic for Applications, virus writers have been able to create viruses that are capable of cross-infecting between the different applications that are part of Office 97. Shiver [76] is capable of infecting both Word 97 documents and Excel 97 spreadsheets. Triplicate [77] is capable of infecting Word 97 documents, Excel 97 spreadsheets and PowerPoint 97 presentations.

\(^{15}\)According to prevalence tables published in *Virus Bulletin*.

\(^{16}\)File-based viruses or worms that utilise network distribution methods such as e-mail also made a strong showing in 1999. These viruses or worms include ExploreZip [74] and Ska [75].
There even exists at least one virus, Beast [78], that is capable of cross-infecting between the Windows environment and Word 97 documents. The virus consists of two parts: a VBA macro and an ordinary Windows executable. When an infected Word document is opened, the virus’ VBA macro installs and executes its Windows executable component. The Windows executable component of the virus checks for uninfected active Word documents periodically, infecting any that become available. Infection of Word documents takes place using facilities provided by Word and Windows. Although the virus does not attempt to infect Windows executables, it does not require much imagination to see that this would be a direction in which this style of attack could be extended.

Office 97 applications will warn the user when documents containing customizations such as macros are being opened, and permit these customizations to be disabled. However, the warning may be easily turned off, and in any case does not really assist the user in determining if a document is infected.

Office 2000 supports low, medium and high security modes to determine the acceptability of macros. Macros are either always acceptable, acceptable subject to user authorization, or never acceptable. However, support for digitally signed macros is also included; signed macros from a source that has previously been designated by the user as being “trusted” are always acceptable and will be enabled without warning [79, 80].

Finally, it should be pointed out that macro viruses exist for a variety of other application packages, as can be seen by browsing a list of known macro viruses such as [73]. However, macro viruses capable of infecting Word documents, Excel spreadsheets or PowerPoint presentations are by far the most common varieties of macro virus currently known.

2.4.4 Polymorphism and Stealth

2.4.4.1 Polymorphism

An easy way to identify a computer virus infecting an executable object such as a file or a boot sector is to search the object for a sequence of bytes found only in the code of that particular virus and in no non-viral code. The sequence of bytes is known as the virus’ signature.
How could a virus attempt to hide its signature so it is more difficult to detect? Well, one strategy is to encrypt the body of the virus using a variable key for different infections. That way, the only constant piece of viral code from which a signature can be extracted is the small routine responsible for decrypting the main body of the virus.

The encryption routine does not need to be particularly sophisticated: its sole aim is to alter the appearance of the virus' code. As the key used to encrypt the code must be available to enable the subsequent decryption of the code, no secrecy is provided.

A more complicated strategy is to not only encrypt using a variable key, but to use a variety of different decryption routines. Now the virus may only be characterised by a small set of signatures, one for each of its decryption routines.

The most sophisticated strategies that have been developed to alter the appearance of a virus' code is to use encryption with a variable key, and also manipulate the decryption routine in such a way that there are a large number of possible decryption routines that all differ in appearance but ultimately perform the same task, that of decrypting the virus' code. If done well, there will be a very large number of decryption routines that may be generated, and it will be impossible to pick a scan string to identify the virus on the basis of its decryption routine. This strategy has come to be known as polymorphism [15, 81]. Viruses that employ a strategy such as this are termed polymorphic viruses.

Some strategies for altering the appearance of code that can be used by a virus writer to implement polymorphism include [82][4, pp. 199–215]:

**Instruction Equivalence:**
Replace machine instructions with another from a set of equivalent instructions that achieve the same effect.

**Instruction Reordering:**
Rearrange a sequence of independent or semi-independent instructions into a different order which still results in equivalent operation.

**Garbage Insertion:**
Insert instructions into a sequence of instructions that have no effect on the outcome of the sequence of instructions. An example would be the insertion of NOP (No Operation) instructions, or instructions that perform operations on registers
that are not otherwise used in the sequence of instructions.

**Build and Execute:**
Include code to construct, somewhere in memory, the code that will perform a particular task rather than writing code to perform that task directly. Once built, execute the code to perform the required task.

**Intermixing Operations:**
Interleave the individual instructions from two sequences of code that perform independent tasks such that the two tasks occur simultaneously rather than in sequence.

Note that these strategies could also be applied to the main body of the virus code instead of using some form of encryption to conceal it.

An interesting article discussing the encryption methods used by various early viruses is [83]. Some of these viruses use strategies such as those listed above to implement polymorphism.

A variety of modules have been created that can be used by a virus writer to make any virus polymorphic. So a virus writer is in theory able to create difficult-to-detect polymorphic viruses without expending much effort on the polymorphic parts at all. These **polymorphic engines** include MtE, the “Mutation Engine” [84] or “Mutating Engine” [4, p. 51], or NED, the “Nuke Encryption Device”.

However, anti-virus techniques have been developed that can detect viruses using sophisticated polymorphic strategies. Brief descriptions of a number of strategies that have been implemented to detect polymorphic viruses can be found in [85]. Polymorphic viruses in general do not present the same level of difficulty for anti-virus techniques as was once the case.

Forms of polymorphism can also be implemented by macro viruses. The sorts of techniques that can be used include [62, 66]:

- Changing the names of variables used in the virus’ macro code;

- Insertion of WordBasic or VBA operators into the macro code that have no effect on the normal operation of the virus;

- Insertion of comments into the macro code;
• Swapping around the WordBasic or VBA operators used in the virus’ macro code in such a way that the virus’ normal operation is not affected.

Many of these strategies are similar to the ones listed above that may be applied directly to a virus’ binary code.

Encryption within a macro virus can be achieved by treating the majority of the virus code as a text string which can be encrypted and inserted into the virus as comment with an associated decryption loop. This strategy is used by the Quoter macro virus [86].

2.4.4.2 Stealth

Many viruses, when active in a computer system, will attempt to conceal signs of their presence in infected executable objects. A virus that employs such techniques has come to be referred to as a **stealth virus** [4, pp. 51–52] [84].

As a simple example, a boot sector infecting virus active in memory might attempt to conceal signs of its presence on infected disks. This can be done by intercepting requests to read from an area of the disk, such as the boot sector, that has been modified by the virus. Instead of returning a boot sector infected by the virus, return the original contents of the boot sector before the disk was infected.

This strategy is most straightforward for boot sector viruses, as these viruses modify a small number of sites on an infected disk. The Brain virus, one of the very first viruses for the IBM-PC, was a boot sector virus that employed techniques as described above [2, 3], and many other boot sector viruses have since employed similar techniques. Stealth is more complicated for file infecting viruses, as the viral code is not found in one convenient place on disk.

If the virus is active in memory, reading an infected object will not reveal signs of the virus’ presence. The most reliable way to detect the presence of a stealth virus is to boot the computer from a disk containing the operating system and that is known to be uninfected. This ensures that the virus will not be active in memory when attempts are made to search executable objects for signs of infection.
2.5 Potential Damage by Computer Viruses

The damage that can be inflicted by a computer virus is limited only by the virus writer’s imagination, and the permissions of the user that is unknowingly running the virus. This statement is not particularly useful for illustrating the damage that can be inflicted by a computer virus, however. To illustrate this, some types of damage that can be inflicted by computer viruses will be discussed. We’ll also look briefly at some of the computer viruses and worms that received considerable media attention in 1999, as a result of a number of high-profile incidents.

A useful discussion of types of damage that can be inflicted by a computer virus can be found in [87]. Arguments against benevolent viruses are presented in [88]; many of these arguments arise as a result of types of damage that can be performed by a computer virus.

Computer viruses cause damage to a computer system in a variety of ways. This may occur as a result of the infection of an executable object — for example, an overwriting virus will destroy part of the executable object it infects. Or the “damage” may be that the presence of the virus on a computer system causes system instability or degradation of performance of a computer system, both possibilities resulting in loss of productivity of affected workers, and the need to divert resources into the removal of the virus and restoration of normal operations. Damage can also occur as a result of bugs, or programmer errors, in the virus. For example, the Jerusalem virus fails to mark .EXE files it infects appropriately, and so infects them multiple times. Infected files can become so extended in length by repeated infections with this virus that they become too large to fit into available memory [3].

More intangibly, passing along a computer virus infection could result in a loss of goodwill towards the company or person responsible.

Computer viruses and worms have received some media attention in 1999, as a result of a number of high-profile incidents. The viruses or worms responsible were:

Melissa Word Macro virus [89, 90, 91, 92, 93, 94, 95]:

The most notable feature of this virus is that when first executed on a computer system it will use certain versions of the e-mail program Microsoft Outlook to e-mail infected documents to the first 50 e-mail addresses in each Outlook address book. The virus can spread so rapidly that mail servers may be overloaded with
huge volumes of Melissa-generated e-mail.

**Chernobyl/CIH virus [27, 96, 97]:**

The most interesting feature of the CIH virus is that it is capable of rendering certain types of PCs unbootable when its damage routine is triggered. It does this by erasing some of the code in ROM required to boot the computer. Normally it is impossible to modify the contents of a ROM chip. However, many computers now incorporate ROMs that can be reprogrammed if necessary, making it easier to update important routines stored in these reprogrammable ROMs. The CIH virus, by erasing code required to boot the computer, renders the computer effectively unusable pending repair. It is easy to see that this could result in considerable expense and inconvenience.

**ExploreZip worm [74, 98, 99, 100]:**

A standalone piece of malicious software, ExploreZip propagates either by sending e-mail containing the worm program in reply to new e-mail received by an infected computer, or via file sharing. It inflicts damage on a system by attempting to truncate certain varieties of file, such as those belonging to Microsoft Office programs, to zero length.

Viruses that inflict massive, immediate damage on a computer system, perhaps by wiping disks or in the manner of the CIH virus, outlined above, are, in fact, less of a threat than viruses which perform more subtle varieties of damage. This is because the damage done makes the presence of the virus obvious and limits future opportunities to spread to other computers. However, such viruses can be very inconvenient to those to whom damage is done.

A data-diddling virus [101] performs its damage by altering data files. These changes can be difficult to detect, as data files change legitimately on a frequent basis. Damaged files may be incorporated into backups; older backups with undamaged earlier versions of files may be discarded or recycled. Therefore, recovering from such a virus attack can be difficult. This style of attack has been implemented by some viruses, for example:

- Nomenklatura swaps the order in a disk's file allocation table of clusters making up large files. If this has been happening for some time, reconstructing the
original ordering of a file's clusters can be difficult [87].

- Variants of the Wazzu macro virus will move up to 3 words in a document being opened to some other place in the document, or insert the word 'wazzu' at some point in the document [102].

A virus could also be designed to mount an extortion-based attack, by using cryptographic techniques. Such a virus could encrypt some of a user's important files, such as those used to store electronic cash, in such a way that the user is unable to recover the encrypted material without either a recent backup or the assistance of the virus writer [103].

Viruses could also be employed as an information-gathering mechanism. Passwords would be a useful target. Again, cryptographic techniques can be used not only to render the information useless to someone besides the virus writer, but also to conceal the quantity of information collected [104].

It has been proposed that computer viruses could be used to perform a distributed computation covertly [105]. Problems faced by such a virus are the distribution of work and the reporting of any results found to the architect of the scheme. Some problems, such as performing a search for the key used to encrypt a block of data using a cryptosystem such as the Data Encryption Standard (DES), can be undertaken by randomly choosing a work element. However, as for reporting results found, the best that can sometimes be done is for future generations of the virus to carry along results found so far, in the hope that at some time in the future, these results will find their way back to the sender of the virus. The increasing popularity of the Internet can offer a solution to both the distribution of work and the return of results: a virus could use the network, when available, to achieve both tasks. In Chapter 5 some non-viral malicious software that performs covert distributed computing and that uses the Internet for the distribution of its code and for fetching work and reporting results will be discussed. In Chapters 7 and 8, some problems are discussed that can benefit from an approach using distributed computing and that are not well-suited to processing by picking a random work element.

Clearly, because of the almost unlimited potential for damage possessed by computer viruses, and the speed with which viruses are able to spread in the modern networked environment using mechanisms such as e-mail, it is important that viruses be
identified and removed as rapidly as possible. There are a variety of strategies that can be pursued to help prevent virus infection, or to detect the presence of viruses.

2.6 Anti-virus Types and Strategies

It would seem to be desirable to either prevent viruses from becoming established in a computer system, or to detect and remove them as soon as possible should the system become infected. How can this be achieved?

There are a variety of non-technical solutions, for example:

- Boot sector viruses are a significant problem for users of IBM PCs. Systems are commonly infected by leaving an infected floppy disk in the floppy drive at system startup time; frequently the default startup sequence is to attempt to startup from the floppy drive, if a disk is present, which results in viral code being executed. On many systems, this can be prevented by changing the order in which drives are searched for a bootable device to check the hard drive first, rather than the floppy drive.

- One way that macro viruses spread is by the exchange of infected documents in e-mail. Some e-mail readers may be configured to automatically open received documents with an appropriate application; if this is configured to be Microsoft Word, any viral macros present in the document may be executed when the document is automatically opened. Instead, configure such programs to use something like the Microsoft Word Viewer to open documents received in e-mail. A variety of other useful techniques for dealing with macro viruses can be found in [106], or alternatively [107, 108, 109, 110].

- Exercise caution in deciding to execute executable programs that have been received from untrusted sources. For example, it is not unknown to receive small executable programs from friends or colleagues that perform some amusing or interesting function. Such programs can also function well as a carrier of a virus from one computer system to another, or may be a trojan horse program that could, for example, install a backdoor into the computer system which can later be used to gain unauthorized access.
• It is important to keep up with patches, security related and otherwise, released for any operating systems and application packages in use. Flaws in operating system or application software can facilitate access of malicious code to a computer system. Due to the complexity of modern software, it would seem that problems such as this are inevitable. For example, BubbleBoy [111] is a virus/worm written in Visual Basic Script which is spread via e-mail and is executed as a result merely of reading an e-mail message, without explicitly executing any attachments. This is possible because of an operating system flaw which has been patched. Following application of this patch, a system is no longer vulnerable to an attack utilising that particular flaw.

• Good backups are important in recovering from a computer virus infection. This includes keeping a backup of the Master Boot Record (MBR) and boot sector, both of which may be modified by a boot sector virus or multipartite virus. By themselves, however, backups are not the answer:

  – In Section 2.5, we have seen that there are existing viruses that are designed to corrupt files in such a way that corrupt versions of the files will be backed up, possibly replacing uncorrupted versions;

  – Backups may not be particularly useful in dealing with a macro virus. This is because documents, such as Microsoft Word documents, commonly change frequently. So the virus cannot be eradicated by replacing infected documents from a backup;

  – There are also no guarantees that backed-up executable objects are not infected by a virus [112]; if a virus has been present in a computer system for a sufficiently long time before being detected, backed-up executables may also be infected. How should a backed-up executable be determined to be virus-free?

More useful techniques to reduce the threat posed by computer viruses can be found in [113].

There has, however, been considerable effort expended in developing anti-virus software to detect the presence of viruses either before or after the computer system has
become infected. In many cases, these viruses may even be removed — the infected program is "disinfected" — to restore the infected program to its original state.

Recall from Section 2.2, however, that it is not possible to detect 100% of viruses. So these anti-virus techniques will not be perfect — they will suffer from false positives, which occur when a file is identified as being infected by a virus when in fact it is clean, or false negatives, which occur when a file is declared to be clean when in fact it is infected with a virus. However, with sufficient work on the part of the developers of anti-virus software, the rate of false positives and false negatives can be made very low. Users of these products must also invest time, energy and money on an ongoing basis in ensuring that any updates made available by the product's developers are installed in a timely fashion; use of outdated security products can contribute to a false sense of security and leave the user vulnerable to new attacks.

The common types of anti-virus techniques are as follows:

- Scanner;

- Integrity Checker; and

- Behaviour Blocker/Activity Monitor.

A good defense can be achieved by a combination of these anti-virus techniques, the goal being to make the creation of a virus that is able to avoid all the implemented defenses as difficult as possible [114].

We also consider briefly possible future developments in anti-virus technologies.

### 2.6.1 Scanner

A virus scanner attempts to detect the presence of a virus in a file by searching for the signature of that virus in the file. The signature of a virus is a sequence of bytes, not necessarily contiguous, that are thought to characterise that virus — the signature is unlikely to occur in a normal, uninfected executable file, so finding it is then a reasonable indication that the file is infected with the virus corresponding to the signature.

If possible, it is useful to use several different virus scanners to confirm a virus infection, should one be detected. This can help to reduce the significance of false positive alerts from a scanner.
2.6. Anti-virus Types and Strategies

Scanners are able to exactly identify a virus, by computing a checksum or hash value of the invariant portions of the virus' body, for example. It may then be possible for the scanner to disinfect the infected files. However, as we have seen, it is possible to write viruses in such a way that the original state of the file cannot be restored. The most reliable way to remove a virus from an executable object is to replace the executable object from a backup if possible.

There are, however, a number of problems with this approach to virus detection [4]:

1. A scanner can only detect viruses for which it has a signature; it cannot detect viruses that were unknown when the scanner, or its virus signature collection, was released.

2. New viruses are constantly being written. To be able to detect new viruses, scanners must be frequently updated. Occasionally, updating a scanner will require the acquisition of a new version of the scanner program rather than just the scanner's data files.

3. Scanners are not good at detecting polymorphic viruses. Polymorphism is discussed in Section 2.4.4.1. Such viruses attempt to minimise the number of bytes that can be used to form a signature for that virus.

4. Scanners require a certain level of discipline from the users. Users must run the scanner to check disks, such as floppy disks from other users, for infection at regular intervals. Executable programs or documents that might potentially contain a macro virus and that have been downloaded from the Internet or received in e-mail are potential sources of infection and should be checked for viruses before use.

Solutions to some of these problems have been devised.

Problem #1 prompted the development of heuristic analysis [115, 116]. Heuristics in this case are a set of rules that can be applied to executable code to determine the possibility that the executable code contains a virus. Heuristics can be both positive — the presence of code to perform a particular function contributes to the belief that the executable code contains a virus — and negative — the presence of code to perform certain functions, like interacting with the user or displaying windows on the screen,
detracts from the belief that the executable code contains a virus. Good results can be obtained using heuristic analysis; however, it is likely that a number of false positives and false negatives will also occur. There are no guarantees that a particular unknown virus will be detected by heuristic analysis.

Once information on the behaviours exhibited by a program has been collected, this information must still be interpreted to conclude if there is enough evidence to declare that code is likely to be a virus. Artificial intelligence techniques may be of use: for example, neural networks have been applied with some success to the detection of boot sector infecting viruses [117].

There are several ways in which problem #3 might be addressed. The first possible solution is to produce a custom detection algorithm for each polymorphic virus or polymorphic engine with which a scanner has problems [84]. This is time-consuming and would otherwise seem to be a poor general approach in the long term. A number of other solutions that have been implemented to help detect polymorphic viruses are summarised in [85]. A more general solution is to implement some form of generic decryption [15, 118, 85].

A polymorphic virus must decrypt the main body of its code before execution. The main body of its code is usually static. Ordinary signature scanning or heuristic analysis can be applied to the virus once its main body has been decrypted. Generic decryption attempts to emulate the program under investigation past the point at which the virus code has been decrypted. The decrypted virus code can then be scanned for signatures in the usual way. There are some difficulties in performing generic decryption: the first is deciding when to stop emulating a program under investigation. This is a problem not only because polymorphic viruses will execute different numbers of instructions in the process of decrypting the virus body, but also because some viruses are written so that the virus receives control at some point during the execution of the infected file, rather than immediately on execution. The second problem faced in the implementation of generic decryption is deciding which of the many possible CPU architectures should be used to emulate the program.

Emulating a program’s execution can also be used in the performance of heuristic analysis: rather than examining the code directly, which may be written to obscure the
function that the code performs by using techniques similar to those used by polymorphic viruses but applied to a different end, a heuristic analyser can gather information based on the actions performed by emulated code [116].

A solution to problem #4 is to provide a memory-resident scanner component that is able to scan files or disks for viruses when accessed. So an executable program might be scanned for viruses before being run, or a Word document before being interpreted by Microsoft Word. Users are no longer required to be particularly disciplined in scanning files — instead, files will be scanned when used.

Some possible solutions to problem #2 will be discussed in Section 2.6.5.

The detection and disinfection of macro viruses poses some special problems for anti-virus developers. For example, it is not usually possible to replace the infected object from a backup copy, because the document to which the viral macros are attached may have changed substantially since the last uninfected backup copy was made; it must be disinfected in some way, even if this requires that all macros be removed from an infected document. These problems are discussed in greater detail in [62, 66].

Finally, scanners have the advantage over other forms of anti-virus software that they can be used to identify files infected by a virus before that virus has been executed and has had a chance to spread. Other anti-virus techniques can detect evidence of a virus’ spread or the actions performed by a virus.

### 2.6.2 Integrity Checker

The idea behind an integrity checker is that a virus must change something when it infects a file or disk. So viruses can be detected by monitoring files for changes. This is done by computing some form of checksum or hash value, which is a short string of bytes derived in some manner from the contents of the file in question. The method for computing the checksum or hash value should ensure that any change in the file produces a different checksum or hash value, and that it is hard to forge this value — that it is infeasible to find a modified program such that the hash values of the two programs are identical.

The checksum can then be recomputed at a later date. If the value has changed from its original computation, then the file in question has been changed.

It is important that it be infeasible to find a modified program such that the hash
values or checksums of the original, unmodified program and the modified version are identical. Otherwise, a virus can compute the extra changes required to make the hash value for a modified program match that of the unmodified program [114]. However, as different integrity checking products may use different methods for computing the checksum, such attacks are unlikely to be successful against all products.

However, an integrity checker does not detect viruses — just changes in files. If a file has changed, then this may be the result of it being infected by a virus. The file in question may have been updated by the user; installing updates to software is not an uncommon activity. A file may even have altered as part of the normal functioning of the application package to which it belongs. Only the user is able to determine why a file might have been altered.

Furthermore, an integrity checker is not able to detect viruses in files that were infected before the integrity checker computed checksums for files in its database. It is also not able to determine if a file downloaded from the Internet or received in e-mail is infected with a virus, because in most cases no checksum is available to verify that what was received by the user is really what was originally distributed by the file's sender. Viruses may only be detected once they have had a chance to spread. Unfortunately, this means that these viruses may also have had a chance to do some damage.

Viruses are detected by detecting modification in files containing executable code. So integrity checkers are vulnerable to an attack by a virus that is able to infect executable files without altering any code, unless such attacks have been considered during the implementation of the integrity checker. Companion viruses, discussed in Section 2.4.1.3 use such a strategy. A variety of such attacks against integrity checkers and countermeasures for these are discussed in [35].

Advantages that integrity checkers have over virus scanners include:

- An integrity checker is able to detect unknown viruses. Unknown viruses are detected as a result of the changes made in files as the virus spreads.

- Polymorphic viruses pose no problems to an integrity checker; polymorphism is useful only for altering the appearance of the virus. The virus must still make changes, which can be detected by the integrity checker.
Considerable research has been performed into the nature of the algorithm required to compute the checksum or hash value and other aspects of the integrity checking process. For a sample of material, see [119, 120, 121, 122, 123, 124, 125, 126, 4]. Bontchev [35] is a very detailed discussion of issues, such as virus attacks directed against integrity checkers, relating to integrity checkers and their implementation.

Algorithms to produce hash values are also heavily studied in the cryptographic literature. These algorithms, known as hash functions, may be used in conjunction with a public-key cryptosystem to produce a signature for a message to allow authentication of the sender of a message, or verification that the message has not been tampered with. A key property of a useful hash function is that it is difficult to find two messages with the same hash value. It is also useful if a hash function is fast: able to process data very rapidly to produce a hash value. These properties may also be useful for anti-virus purposes. An example of a well-known hash function is MD5 [127]. More hash functions, and references to further reading, can be found in [128] by a reader interested in pursuing this topic.

Finally, the application of an integrity checking strategy to the detection of macro viruses poses some special problems. As the non-macro part of the document may change frequently, computing the checksum or hash value for an entire document is not useful. The macro areas alone must be examined. There is also a variety of “companion” attack that can be implemented by a macro virus, so that viral macros are not added directly to a document to infect it [62].

2.6.3 Behaviour Blocker/Activity Monitor

This sort of anti-virus program attempts to detect the presence of an active virus by monitoring the computer’s operation for activity that is considered to be indicative of the presence of a virus. For example, it might be considered suspicious to attempt to alter in some way another executable object or some component of the operating system.

Problems with this sort of anti-virus measure include:

- Viruses may make use of operating system implementation weaknesses to accomplish their goals, but it is not required that a virus do so; most of the operations that a virus performs have entirely legitimate applications.
• Some viruses will attempt to bypass operating system patches installed by a behaviour blocker to monitor system activity. Some Macintosh viruses, for example, contain code to attempt to redirect traps called by the virus to ROM code on certain machines. If successful, the behaviour blocker will fail to alert even though the virus is active.

• Blocking a suspicious action can have unexpected consequences. If the action is not being performed by a virus, or if the action is being performed by a virus but is part of an already partly-completed sequence of actions, then blocking the action may leave files in an inconsistent, damaged state.

• If an action is considered to be suspicious, it is not uncommon to leave the decision over whether or not to permit the action to take place to the user of the software. This can be irritating if it happens frequently not as a result of the actions of a virus, generating many false positives, but also may require a reasonably detailed knowledge of the current operating environment to determine whether or not an action is permissible.

2.6.4 Other Anti-Virus Techniques

There are a variety of other anti-virus techniques that have been suggested but that see little use in practice, for a variety of reasons. These techniques include use of snapshots during the computer bootstrap process [114, 129] [4, pp. 77–78] and vaccines [4, pp. 71–72].

A snapshot works by making a copy of a computer’s memory at some stage during the bootstrap process at a time when the system is presumed to be known to not be infected with a virus that modifies the bootstrap process — some sort of boot sector infecting virus. On subsequent booting of the computer, the state of memory at the same point in the bootstrap process at which the snapshot was originally taken is replaced with the state recorded in the snapshot. Any boot sector infecting viruses that have already become memory resident are removed when the snapshot is used to replace the contents of memory, and the bootstrap process can continue from a known clean state.

Unfortunately, the use of a snapshot complicates modifying the configuration of
2.6. Anti-virus Types and Strategies

the operating system: for example, installing software which loads during the boot process at a point before the snapshot was taken requires that the snapshot be replaced with a new snapshot incorporating the newly-installed software.

The idea of a vaccine is to make modifications to executable programs in such a way that those programs will be recognised by a virus as already being infected by that virus. This technique has many problems, however:

- There are a very large number of viruses against which a program must be vaccinated. Attempting to vaccinate against more than a small subset may make a program inoperable;

- Not all viruses attempt to determine if a program is already infected before attempting to infect it; these viruses cannot be vaccinated against;

- Mutually exclusive conditions employed by some viruses may make vaccination impossible. As a simple example, if virus A decides that it has infected a file if byte $L \leq 75$ and virus B decides that it has infected a file if byte $L > 100$, it is impossible to vaccinate against both these viruses.

2.6.5 Future Developments

As has been seen recently with the Melissa virus, the Internet enables the rapid transmission of viruses from one site to another. A virus can become widespread so rapidly that traditional anti-virus techniques have difficulty coping. For example, the viruses detected most reliably by virus scanner, discussed in Section 2.6.1, are those whose signature is present in the scanner's database of virus signatures.

Such problems can be partially addressed by creating anti-virus tools able to update themselves across the Internet, either automatically or at the request of the user. This can significantly reduce the burden on the user of manually frequently updating anti-virus tools. Improved tools for distributing anti-virus updates to users on a local area network in such a way that these updates are largely transparent to the user can also help.

However, this still requires that new viruses be analysed, signatures produced and anti-virus tools such as scanners appropriately updated. These procedures may take too long to effectively respond to fast-spreading viruses.
2.6. Anti-virus Types and Strategies

What more can be done? Considerable research has been undertaken into producing an anti-virus tool, a “computer immune system” [130, 131], that is capable of:

- Detecting the presence of viruses previously unknown to that particular version of the anti-virus tool;

- Identifying a signature for previously detected unknown viruses, possibly with the assistance of a remote site where virus samples identified by a client computer are automatically analysed and a signature produced;

- Distribution of signature for newly-identified virus to other clients requiring detection for this virus.

Ideally, human involvement in this process would be minimised so that response times will be short. For example, potential virus samples might be dispatched automatically for automated analysis at a remote site, or perhaps with the permission of a human operator, who might ensure that proprietary materials involved in a virus incident do not leave a company’s site.
Chapter 3

Companion Viruses and the Macintosh: Threats and Countermeasures

The material in this chapter is based upon a published paper:


Much of the introductory material present in the paper on which this chapter is based has been edited out to avoid unnecessary repetition of material already covered in Chapter 2.

3.1 Introduction

The Macintosh virus world is not as active as that of PC and PC-compatible computers. Macro viruses are as much a problem for users of affected application packages on the Macintosh as on the PC, but there are only a few dozen known native Macintosh viruses. This number has seen little change in some years. To the best of our knowledge, the techniques described here are not employed by any existing Macintosh virus.

A “companion virus” is a variety of computer virus which avoids modifying the files that it “infects”. Companion virus attacks that can be implemented when running under DOS are discussed in Section 2.4.1.3. The first method of attack is based on the order of execution of files when a file extension is not specified to differentiate between two files of the same name and different extension, and the second on placing
3.1. Introduction

an executable of the same name and same extension earlier in the search path followed by the operating system when identifying which executable to run. As a virus of this type does not modify the application program which it infects, it is an attack that is targeted specifically at anti-virus defenses based on integrity checking, a type of anti-virus method that endeavours to detect changes in executable files that take place as a result of infection by a virus. Integrity checking is discussed in Section 2.6.2. Integrity checkers must be modified to be able to detect the presence of a companion virus in the environment, by looking for changes not only in an application program, but also in that program’s environment.

However, the Macintosh operating system does not have the concept of a “path” which is followed when searching for executable files, or the notion of filename extensions. However, several features of the Macintosh operating system enable an attack that appears very similar in conception if not in execution.

We consider this results in a minor weakness in the Macintosh operating system, which could potentially be exploited to construct a companion-type virus. This type of virus has been previously unknown on the Macintosh platform. For detailed information on strategies that have been employed previously to implement viruses on the Macintosh, consult Section 2.4.2. We decided to explore the consequences of this attack in more detail, and to attempt to devise some countermeasures that could be implemented by anti-virus software to enhance its effectiveness against a virus employing these techniques. The strategies involved in implementing a companion-type virus on the Macintosh have been tested using versions of the Macintosh operating system from 7.1 to 8.1, and may also be effective on later versions of the operating system. Macintosh users and anti-virus vendors should be aware of the possibilities we outline. We discuss details of the attack and some thoughts on detecting and countering the attack in Section 3.3.

In the interests of not supplying sufficient information to allow the easy implementation of a functioning virus, only the basic ideas behind the attack will be discussed. Source code will not be supplied. We feel that the security community is better able to respond to a potential security problem of this nature with some degree of foreknowledge of the problem.
3.2 More Macintosh Basics

3.2.1 The Desktop Database

Recall from Section 2.4.2 that every file maintained on disk by the Macintosh operating system is associated with both a file type and a file creator.

The file creator is used to associate the file with the application program that created it. This allows the correct application to be started when a file is double-clicked by the computer user. The type information may be used by the application to determine how the contents of the file should be interpreted.

The Macintosh operating system identifies the application to be executed in response to a user double-clicking a file by searching a database in which the location of each application with a valid creator code is maintained. This is not the only information stored in this database — icon information for the different file types created by an application is also stored in the database, if this information is provided by the application. The database is known as the “Desktop Database”.

Not every type of disk has a Desktop Database. Volumes storing less than 2 megabytes, such as floppy disks, do not have a Desktop Database, but instead have a simpler structure which performs similar functions. Our attention here is confined to disks which do have Desktop Databases, such as hard disks, or for which one is created, such as AppleShare volumes [42, pp. 9-3–9-4].

Usually there will be a single application with a certain creator code on a disk. However, it is possible to have several applications with the same creator code on the one disk. Although according to a technical note [132] the application which is the “first choice” is the one with whose information the Desktop Database was last updated, correcting earlier documentation stating that the “first choice” application was the one with the most recent creation date [42, p. 9-5], it appears that in most cases the application selected will be the one with the most recent creation date. Even after rebuilding the Desktop Database, a process that may be initiated by the user and which is sometimes useful in troubleshooting, the application selected is the one with the most recent creation date. It is critical to the attack described in Section 3.3 that the application selected for execution be the viral application.
3.2.2 Starting an Application

When the user starts an application by opening a file or files in some manner, the application is notified by the operating system of the files that were selected by the user.

This is accomplished using an Apple Event, which is a type of high-level event commonly used for interapplication communication. There are many different types of Apple Event, and applications may define their own. In the case being considered here, the operating system sends an “Open Document” (odoc) Apple Event to the application when it has started executing to inform the application of the location of the files that the user wishes to open using that application.

Not all application programs support receiving Apple Events; a program that does not support these events is not a candidate for infection by the method we will describe. Applications may provide Apple Event support but don’t themselves “own” any files; as will be seen, such applications are not good candidates for infection. However, many common application programs that create files that are “owned” by that application support these events, and would be more difficult for users to work with if they did not.

3.3 A Macintosh “Companion Virus”

How might these facilities provided by the operating system be used to implement a viral attack? If the application required is not already running and is not specified, only the documents that are to be processed, then the operating system must identify and execute the appropriate application(s) itself, passing an event to the application to inform it of the documents it is to process. When there exists more than one application on a given disk with the particular creator code, the operating system selects the one with the most recent creation date.

So it suffices to infect an application by creating an application program with the same creator code as the other application that is the target of infection, but with a more recent creation date, such that the operating system executes the viral application in preference to the application which is the target of infection.
Then, when an infected application is launched, to make it seem as though everything is normal, the viral application performs the following tasks:

1. Intercepts the event intended for the infected application which is sent by the operating system; and

2. Runs the infected application and forwards it the intercepted event; soon after completing this step, the viral application would exit, to avoid easy detection.

There are a number of other details which must be handled to create an effective virus. As an example of these details, it is useful to preserve the original application icon information, which is usually overridden by the icons applicable to the more recently created application, so that visual displays look unchanged. We are reluctant to discuss solutions to such problems here, in the interests of not revealing enough information to easily create an effective virus.

It is possible to extend the implementation to handle Apple Events sent to a newly-launched application under other circumstances and that require different handling. However, the presence of the viral application can result in behaviour that differs slightly from that of the pre-infection state.

Cooperating applications might use other types of high-level events that are not Apple Events. We consider this to be sufficiently uncommon that the possibility is not addressed here.

Clearly this attack bears some resemblance to a companion virus style of attack as described in Section 2.4.1.3.

It should be noted that it is certainly possible to specify exactly which application program is to be used to manipulate a particular file or files; such a virus would rely on the fact that it is more convenient to permit the operating system to identify and run an application than to perform this task manually.

Some consideration has been given to how such a virus might become resident in memory; that is, how it might place viral code somewhere in memory and arrange for it to be executed at some time in the future, long after the viral application itself has ceased to run. Installing a device driver is one possible option. Device drivers are not required to deal with devices at all; other uses have been found for them. A device driver may receive calls from the operating system to perform periodic tasks — in the
3.3. A Macintosh "Companion Virus"

case of a companion virus, such a periodic task could be searching the list of currently running application programs, infecting any that appear to be uninfected.

3.3.1 Detection

The mere presence of several applications with the same creator code on a Macintosh computer system is not something which should cause any alarm, and is not sufficient to conclude that an application program has been infected by a companion-type virus. This situation commonly arises when, for example, a new version of an application package is installed without removing the previous version. An integrity checker would need to monitor other information to help it decide how alarming the presence of multiple applications with the same creator code is. For example, as a companion virus would most likely be a much smaller file and of a simpler structure than an application that has some more useful functionality, the presence of two application programs with the same creator code but very different sizes or structures might be considered suspicious.

It would be useful for an integrity checker to keep track of the locations of legitimate applications, and require authorisation from the user to recognise a new or moved application. An integrity checker that is aware of legitimately installed applications can consult the Desktop Database for a specific creator code to determine if any new applications have been added.

There are a number of system calls which may be of use in implementing such a virus and which may be considered suspicious by a behaviour monitoring program. A behaviour that is a characteristic of such a virus is a need to launch the infected application. As the viral application and infected application have the same creator codes, if a patch were installed on the operating system routine responsible for launching an application, it could check the creator code of the application originating the request against the creator code of the application being launched, and refuse to launch an application having the same creator code as the one making the request; an indication to the user that suspicious activity has been detected would certainly be appropriate. There are various ways that might be used by a virus to circumvent such a check, but employing this check cuts off the simplest and most straightforward way for one application to launch another application. An occasion where one application might
legitimately need to launch another with the same creator code seems most unlikely.

As the virus exists as an application separate to the infected application, checks on
the creation of applications may also be effective. Some anti-virus applications will
likely include such checks, at least as an option, as it may be effective against other
varieties of virus. However, this is not as specifically targeted against a companion
virus attack as the previous countermeasure, and would not seem to be appropriate
to as wide an environment — for example, people working with compilers may find
checks on creation of application programs produce many false alarms.

Under OS 8.0 and 8.1, application files may be marked as “invisible” or be located
within invisible directories, and still be executed successfully when a document is
double clicked. Under most earlier versions of the operating system, this is not the
case\(^1\). Although under some OS versions applications may be hidden, as there would
seem to be no good reasons for this, the presence of such concealed applications could
even be seen by an anti-virus program as being suspicious. Substituting a non-hidden
application if available for a hidden one at time of execution is a potentially useful
strategy. Fixing the operating system to ignore hidden applications would also be a
useful strategy.

The possible utility that a device driver might have for a companion virus is dis-
cussed in Section 3.3. There are many legitimate reasons that a program might wish
to install a device driver\(^2\). As a device driver is potentially of use in a virus attack,
it would be useful to check drivers for suspicious code that might perform virus-like
actions when installed.

Other calls by the virus may also be able to be monitored. For example, the com-
panion virus might use the\footnotesize:\texttt{PBDTA\textbackslash d\textbackslash d\textbackslash APPL}\textnormal{\large} call to update the Desktop Database with
information about the newly created viral application when performing an infection,
rather than waiting for the operating system to update the Desktop Database with in-
formation about the new application at some time in the future. Apple documentation
discourages the practice of making modifications to the Desktop Database [42, p. 9-3],

\(^1\)Sometimes invisible applications will be launched under earlier versions of the operating system
where this is not normally the case, but apparently only just after the application has been made invisible.
This appears to be due to stale catalog information being read from the disk; this behaviour passes
quickly.

\(^2\)For example, they are commonly used to implement virtual disk schemes, where the raw disk data
resides in a large container file.
so it might also be viewed as a suspicious event.

### 3.3.2 Demonstration Program

A non-viral application program demonstrating this attack has been created. The demonstration program has been found to work appropriately in a variety of environments — single and multiple partition Macintosh hard drives, removable media such as Zip disks, and a simple network consisting of two Macintosh computers. The code can be made available to serious researchers upon request.

### 3.3.3 Dangers Posed?

Having discussed a method by which a companion virus for the Macintosh might be written, some consideration ought to be given to the dangers posed by an attack of this nature.

It is more difficult for a virus constructed in this manner to remain undetected and to spread between systems. Under most OS versions prior to 8.0, this variety of virus must exist as a distinct non-invisible file on the disk if it is to be executed, and is noticeable by an observant user. Furthermore, in the interests of surviving to multiply, such a virus if visible would place itself somewhere not associated with the infected application, and so would be unlikely to spread through distribution of software archives.

Under OS versions after and including 8.0, this variety of virus has more options. It could even conceal itself in the same directory as the application which is the target of infection, which enhances its chances of distribution via software archive.

Such a virus may be able to spread via a local area network to another Macintosh. Its ability to spread across a network could of course be slowed by proper configuration of user permissions. In particular, users should not have permission to make changes on network volumes unless absolutely necessary.

The attack is rendered considerably more potent if the virus is able to become resident in memory. One way that this might be accomplished is outlined.

The attack is perhaps most dangerous if the virus so constructed is capable of two modes of infection. For example, one time in ten the virus might infect by modifying
the target program in the manner of a file-infecting virus; although it would be readily detected by an integrity checker, and perhaps by a behaviour monitor, this would improve its chances of spreading to another computer. The undetected copies of the virus which infect using the “companion” strategy would form a reservoir for future infections.

3.4 Conclusion

We have considered a possible virus attack that could be implemented under the Macintosh operating system. The attack has a good resemblance to a companion virus style of attack.

We know of no Macintosh viruses implementing an attack such as is described here. The attack is not believed to pose as great a danger as other varieties of computer virus, due to limitations of the implementation described. It could, however, avoid detection by an integrity checking program or other generic anti-virus measures that were not aware of the possibility of this implementation of the companion virus strategy.

A similar attack could be implemented by replacing the application to be infected with a viral application and renaming the original application\(^3\); we do not consider this attack to be as interesting as the one described in this chapter, because the original application is modified by being renamed.

We discuss various countermeasures that might be employed against such a virus as has been described in this chapter. We believe that the most effective of these measures is to check the creator code of the application attempting to launch another against that of the application being launched, and to abort the request if the creator codes match. Infection is not prevented, but is readily detected.

\(^3\)This appears to be how the \textit{CODE 9811} virus operates [133].
Chapter 4

Extensions to the Macintosh Companion Virus Attack and Some Thoughts on the Future of Viruses on the Macintosh

4.1 Introduction

In Chapter 3, we discussed how a companion virus attack might be implemented on a computer running the Macintosh operating system. The attack used the observation that an application program with the same creator code as another application but a more recent creation date will in some cases be executed by the Macintosh operating system in place of the other application. The viral application was then able to launch the infected application so that the user would have difficulty detecting the presence of the viral application. The process of infection did not require the alteration of the application to be infected in any way.

A simple countermeasure was discussed that worked by patching the operating system’s LaunchApplication routine; the patch would be able to detect an application launching another with the same creator code, and flag this as possible suspect activity.

In Section 4.2, we consider some extensions to the basic attack that can allow the a Macintosh companion virus to avoid detection via a patch on LaunchApplication. These extensions require only the presence of what are now standard operating system services to accomplish their objective.

The Macintosh viruses that have been seen so far and that achieve infection of an application by altering that application’s code have only attacked the 680x0 (68K) portions of the application; any code intended for execution on Macintosh computers based on PowerPC microprocessors has not been touched. As 68K executable code
will not ordinarily be executed if PowerPC executable code is present, this very much limits the virus’ ability to spread successfully. Why have no viruses been seen which attack an application’s PowerPC executable code? In Section 4.3, we consider the level of difficulty faced in the design and implementation of such a virus.

Finally, in Section 4.4, we consider the effects that changes in the Macintosh operating system may have on future viruses for the Macintosh.

4.2 Extensions to the Macintosh Companion Virus Attack

There are a number of ways in which a Macintosh companion virus may avoid detection by a patch placed on the LaunchApplication trap. The key to avoiding detection is for the infected application to be launched when another application appears to be the foremost application.

How might this be achieved? One way would be for a viral application to employ a device driver whose sole purpose would be to call LaunchApplication when some application other than the viral application is foremost. However, we will be considering a much simpler method that relies only on standard operating system components. In particular, we build and execute small scripts written in the AppleScript scripting language. These scripts direct the Finder, the component of the Macintosh operating system that is responsible for tasks such as providing a user interface to the file system, to launch the infected application.

AppleScript is a simple scripting language supplied as a standard operating system component since the release of System 7.5, but which was also available as an extension to System 7.1. It allows users to write simple programs to control applications that have been designed by their developers to be scriptable. To be scriptable, a developer must provide a dictionary that can be used to translate the high-level English-like language in which AppleScripts are written into a series of Apple Events that will be sent to the application as a result of executing an AppleScript. Of course, the application must also process the events received and perform the appropriate actions. A single AppleScript might interact with several applications in the process of executing the script. An application does not need to be scriptable to be able to use AppleScript,
4.2. *Extensions to the Macintosh Companion Virus Attack* 73

however; any application may use AppleScripts to perform tasks. The use of scripting facilities from within a program is documented in [134].

So the process followed by a viral application might look something like:

1. Viral application receives a list of documents via an Apple Event that the computer user intended to be opened by the infected application;

2. Viral application constructs a simple AppleScript which instructs the Finder to use the infected application to open the documents received by the viral application in Step 1.

3. Viral application executes AppleScript constructed in Step 2; the Finder receives an Apple Event which results in the infected application being executed, receiving the list of documents originally received by the viral application.

4. Viral application may now exit.

This process allows the viral application to execute the infected application without itself calling `LaunchApplication`. However, the process does have a number of disadvantages:

1. The process of compiling and executing the AppleScript is very much slower than the alternative of calling `LaunchApplication` directly. As a result, the computer user is more likely to notice that something is amiss. It is possible that this problem may be circumvented by constructing the Apple Event to be sent to the Finder manually rather than using an AppleScript to perform this task.

2. The AppleScript approach does not work well if documents are to be printed rather than just opened.

3. The AppleScript approach also does not work if some other type of event other than an "open documents" event is directed at the viral application initially. However, this is also something for which only limited functionality can be achieved using `LaunchApplication` directly.

This enhancement to the basic Macintosh companion virus attack is considered unlikely to be implemented. In view of changes that can be expected to the Macintosh
4.3 Viruses and the PowerPC-based Macintosh

In this section, we will be considering how a virus might attempt to infect an executable program built for a PowerPC-based Macintosh computer. Programs built for a PowerPC Macintosh have a different internal structure from that of a program built for a 680x0-based Macintosh computer; in fact, in many cases PowerPC and 68K code can be combined into a single program file, which is capable of being executed on either a PowerPC or 68K-based Macintosh, the appropriate code component being executed on computers of each architecture. The basic structure of a PowerPC executable program has some similarities to the executable program structure employed by many modern UNIX operating systems, or with modern versions of Microsoft Windows. The structure of PowerPC programs will perhaps be more familiar to those with some knowledge of UNIX and Windows file formats than was that of a 68K program.

A PowerPC Macintosh computer uses a fragment as its basic unit of executable code and data associated with the code. Fragments are managed by an operating system component referred to as the Code Fragment Manager. The PowerPC Macintosh's runtime architecture may also be referred to as a CFM-based runtime architecture. Complete documentation of fragments and the Code Fragment Manager may be found in [48] and [47].

A fragment may be of almost any size, and may be stored in either a file's data fork or in a resource located in the file's resource fork. Before it can be used, a fragment must be prepared by the Code Fragment Manager. A fragment is prepared by locating any shared libraries, also known as dynamically linked libraries (DLLs), that are required to resolve references to code and data not a part of the fragment itself, and replacing these references with the addresses of the actual code and data. A shared library is a fragment which provides code and data for use by another fragment; one shared library fragment may depend on other fragments which will also need preparation. Shared library support means that code used by different application programs
can be present on a computer system only once, and be shared between the different applications that use it, rather than a copy of the library code being present in each application that uses the library.

A file may contain multiple fragments; the Code Fragment Manager expects to find the index to these fragments in a resource, stored in the file’s resource fork, of type `cfrg` and ID 0. This resource should be provided even if only a single fragment is contained within the file.

When an application is executed on a PowerPC Macintosh, the `cfrg` resource of ID 0, if present, will be searched for a fragment containing the application’s code. This fragment is prepared for execution, and its main routine is executed.

How might such an application be infected by a virus? One obvious way is to modify the executable code contained within a fragment. Fragments may have a complex internal structure, however; a simpler way is to add an extra fragment to the application that contains the viral code. This fragment may be placed within a resource or in the application’s data fork, and an entry added to the `cfrg 0` resource to identify the fragment as the fragment to be executed when an application is run. This approach has been tested using a simple non-viral fragment containing code which beeps when executed, then prepares the application’s original fragment for execution and jumps to its main routine. The approach appears to be effective; the very basic test described here was not exceptionally difficult to implement.

There are a number of potential difficulties with this approach, however:

- If an application cannot be executed because the preparation of the fragment containing the application’s code would require a shared library that cannot be located, an error message is displayed by the operating system. Using the simple approach outlined above, the application’s fragment is prepared by the viral code rather than the operating system, so the virus would need to display any errors ordinarily displayed by the operating system in order to remain unnoticed.

- Applications may cease to work properly after infection if any shared libraries required by the viral code are not available at runtime.

- There may be version conflicts between shared libraries required by the viral code and the application code.
It is believed that most of these difficulties can be overcome by keeping the viral code very simple.

A fragment-based PowerPC virus would perhaps face a more difficult task in becoming memory resident and subsequently infecting further applications. Attempting to support both the 68K and PowerPC architecture in a single virus would also add significantly to the complexity of the virus. These problems do not appear insurmountable, however.

A number of recommendations can be made for detecting the activity of such a virus as it attempts to infect an application, or detecting that an infection may have taken place:

- Adding a new fragment to the application will require modification of the cfRG0 resource. Any such modifications could be treated as being suspicious.

- Modification of the code present in an existing fragment will most likely require making a modification to the application’s data fork. Such modifications to a file which also contains a cfRG0 resource should be treated as suspicious.

- While fragments may be stored in a resource located in an application’s resource fork, locating fragments in the data fork has some advantages for the application developer and computer user not shared by locating fragments in a resource, so in practice it would seem unlikely that resource-based fragments would be common in a PowerPC program. Any such occurrences could be treated as suspicious.

So it can be seen that writing a PowerPC-based virus poses a more challenging undertaking to the virus writer than that faced in writing a 68K-based virus. However, the greater challenge would not appear to pose an insurmountable problem. No 68K-based Macintosh computers have been produced since 1995, and although 68K viruses are still able to spread to some degree on a PowerPC-based Macintosh, it would seem likely that PowerPC-native viruses will eventually be produced. Strategies should be considered for dealing with such viruses before their appearance.
4.4 Affect of Operating System Changes on Macintosh Viruses

The Macintosh operating system has changed substantially over the last few years. The next major upgrade to the operating system will be MacOS X, due for release in late 2000 or early 2001. MacOS X will feature some changes that may affect Macintosh-based viruses:

- MacOS X will feature a number of different execution environments, including a “Classic” compatibility environment which will allow applications compiled to 68K or PowerPC code but written for earlier versions of the operating system to be executed by MacOS X, and a “Carbon” environment; programs written to run in the “Carbon” environment will take more advantage of the new features introduced with MacOS X, which will include protected address spaces and improved support for multitasking. Programs written to run in the “Carbon” environment will be referred to as “Carbonised” programs.

- MacOS X will not execute 68K code, which will run only in the “Classic” compatibility environment. This may affect the long-term viability of viruses written using 68K code, as ideally the use of applications requiring the “Classic” compatibility environment will be reduced in favour of “Carbonised” versions of the application.

- Patching traps is not supported for programs written to run in MacOS X’s “Carbon” environment. It remains to be seen if an alternative will be provided which allows similar results to be achieved. Patching traps is one of the mainstays of many currently known viruses. Patching traps can also be used to detect viral activity, as was illustrated by the patch to LaunchApplication that we used to detect the activity of a program implementing the companion virus attack described in Chapter 3. Patching traps has also been used by developers in the past to extend operating system functionality.

- Some viruses make extensive use of overriding or replacing definition procedures, which are commonly stored in resources, that are used by the operating system and applications to present a graphical user interface. There will be some
changes to the way these definition procedures are handled by MacOS X that may affect how and if these definition procedures are utilised in writing viruses.

- We have made some reference to uses to which a virus might put a device driver, which can presently be installed at any time by any application. This variety of device driver and mode of operation is unlikely to be supported in MacOS X.

It is likely that viruses infecting applications that execute within the “Classic” compatibility environment will also execute successfully within that environment, and be able to infect other applications. It is not likely, however, that old, non-MacOS X-aware viruses will be able to infect “Carbonised” applications.

However, despite the changes that are to occur, experience with UNIX-like operating systems and PC operating systems such as Microsoft Windows suggests that methods of implementing viruses will eventually be found by those interested in doing so.
Chapter 5

Covert Distributed Computing Using Java Through Web Spoofing

The material in this chapter has been published as:


5.1 Introduction

There are many problems in computer science which may be solved most easily by the application of brute force. An example of such a problem is the determination of the key used to encrypt a block of data with an algorithm such as DES (Data Encryption Standard). The computer time required could be obtained with the full knowledge and cooperation of the individuals controlling the resources, or covertly without their knowledge by some means. A past suggestion for the covert accomplishment of tasks such as this involved the use of computer viruses to perform distributed computations [105].

An approach involving computer viruses may not be the best solution to the performance of distributed computing. Distributing new problems on which work is to be done is a potential difficulty, as is the collection of results of the computation in a timely manner. It will also be difficult to determine that a complete search has been performed; this may be important for some classes of problems. Other solutions can be considered in the modern highly networked environment.
Java is a general purpose object-oriented programming language introduced in 1995 by Sun Microsystems. It is similar in many ways to C and C++. Programs written in Java may be compiled to a platform-independent bytecode which can be executed on any platform to which the Java runtime system has been ported; the Java bytecodes are commonly simply interpreted, however speed of execution of Java programs can be improved by using a runtime system which translates the bytecodes into native machine instructions at execution time. Such systems, incorporating these Just-In-Time (JIT) compilers, are becoming more common. The Java system includes support for easy use of multiple threads of execution, and network communication at a low level using sockets, or a high level using URL objects [135].

One of the major uses seen so far for Java is the creation of applets to provide executable content for HTML pages on the World Wide Web. Common Web browsers such as Netscape Navigator and Microsoft Internet Explorer include support for downloading and executing Java applets. There are various security restrictions imposed upon applets that are intended to make it safer for users to execute applets from unknown sources on their computers. One such restriction is that applets are usually only allowed to open a network connection to the host from which the applet was downloaded. A number of problems with Java security have been discovered by various researchers [136, 137].

Java could also be applied to performing a distributed computation. Java's straightforward support for networking and multiple threads of execution make construction of an applet to perform the computing tasks simple. The possibility of using Java applets to covertly or otherwise perform a distributed computation is discussed by several researchers [138] [136] [137, pp. 112–114] [139].

There have been no suggestions, however, as to how this might be accomplished without requiring browser users to knowingly visit a particular page or Web server at the beginning or sometime during the course of each session with their Web browser, so that the applet responsible for performing the computation can be loaded. This paper describes how the Web spoofing idea described by the Secure Internet Programming Group at Princeton University can be used to pass a Java applet to perform a distributed computation to a client. The advantage is that clients do not have to knowingly (re)visit a particular site each time, but may rejoin the computation through bookmarks
made during a previous session.

There will be some indications visible in the browser when rejoining a computation through a bookmark that the user has not reached the site they may have been expecting; however, these signs are small, and are believed to be mostly correctable using the same techniques as employed in a vanilla Web Spoofing attack.

We also propose a simple countermeasure against such a spoofing attack, which would be useful to help users detect the presence of Web Spoofing. The countermeasure takes advantage of the requirement that many clients should be able to be handled simultaneously so that the distributed computation proceeds efficiently.

Finally, we introduce the idea of browser users, as clients of Web-based services provided by third parties, “paying” for these services by running a distributed computation applet for a short period of time. This may be an acceptable alternative to the inclusion of advertising material in certain circumstances.

5.2 About Web Spoofing

Web Spoofing was first described briefly by Cohen [140]. The Web Spoofing attack was later discussed in greater detail and elaborated upon by the Secure Internet Programming Group at Princeton University [141]. Among other contributions, the Princeton group introduced the use of JavaScript for the purposes of concealing the operation of the Web Spoofing attack and preventing the browser user from escaping from the spoofed context. JavaScript is a scripting language that is supported by some common Web browsers. JavaScript programs may be embedded in an HTML page, and may be executed when the HTML page is loaded by the browser, or when certain events occur, such as the browser user holding the mouse pointer over a hyperlink on the page.

The main application of Web Spoofing is seen as being surveillance, or perhaps tampering: the attacking server will be able to observe and/or modify the Web traffic between a user being spoofed and some Web server, including any form data entered by the user and the responses of the server. It is pointed out that “secure” connections will not help — the user’s browser has a secure connection with the attacking server, which in turn has a secure connection with the server being spoofed [141].
Web spoofing works as follows: when an attacking server www.attacker.org receives a request for an HTML document, before supplying the document every URL in the document is rewritten to refer to the attacking server instead, but including the original URL in some way — so that the attacking server is able to fetch the document that is actually desired by the browser user. For example,

http://www.altavista.digital.com/

might become something like:


There are other ways in which the spoofed URL may be constructed. The Princeton group gives an example [141].

The first part of the URL (before the ‘?’) specifies a program that will be executed by the server. This is a CGI (Common Gateway Interface) program. For those not familiar with CGI programs, the part of the URL following the ‘?’ represents an argument or set of arguments that is passed to the CGI program specified by the first part of the URL. More than one argument can be passed in this way — arguments are separated by ampersand (‘&’) characters.

So, when the user of the browser clicks on a spoofed URL, the attacking server is contacted. It fetches the HTML document the user wishes to view, using the URL encoded in the spoofed URL, rewrites the URLs in the document, and supplies the modified document to the user. Not all URLs need be rewritten to point at the attacking server, only those which are likely to specify a document containing HTML, which is likely to contain URLs that need to be rewritten. In particular, images do not generally need to be spoofed. However, as many images would be specified using only a partial URL (relative to the URL of the HTML page containing the image’s URL), the URLs would need to be rewritten in full, to point at the appropriate image on the server being spoofed.

There will be some evidence that spoofing is taking place, however. For example, the browser’s status line and location line will display rewritten URLs, which an alert user would notice. The Princeton group claim to have succeeded at concealing such evidence from the user through the use of JavaScript. Using JavaScript to display the proper (non-spoofed) URLs on the status line when the user holds the mouse pointer
5.2. About Web Spoofing

above a hyperlink on the Web page is straightforward in most cases; using JavaScript
to conceal the other evidence of spoofing is less obvious.

In the course of implementing a program to perform spoofing, we have observed
that some pages using JavaScript do not seem amenable to spoofing, especially if the
JavaScript itself directs the browser to load particular Web pages. It seems that the
JavaScript has difficulty constructing appropriate URLs if the current document is be-
ing spoofed, due to the unusual form of the spoofed URLs.

We have implemented only the spoofing component of the attack, as well as the
simplest use of JavaScript for concealment purposes, that of displaying non-spoofed
URLs on the browser status line where necessary, for the purposes of demonstration;
the aspects of the attack that provide more sophisticated forms of concealment were not
implemented. We believe that the work on concealment of the Web Spoofing attack
done by the Princeton group can profitably be applied to concealing the additional
evidence when using Web spoofing to perform a distributed computation.

5.2.1 Some HTML Tags to Modify

Any HTML tag which can include an attribute specifying a URL may potentially re-
quire modification for spoofing to take place. A few of the more common tags that
require modification to undertake a spoofing attack include:

- Hyperlinks, which are generated by the HREF attribute of the <A> tag. A new
  HTML document is fetched and displayed when one of these is selected by the
  user.

- Images displayed on an HTML page are specified using the <IMG> tag. Its
  attributes SRC and LOWSRC, which indicate from where the browser is to fetch
  the image data, may require adjustment if present.

- Forms into which the user may enter data are specified using the <FORM> tag.
  Its ACTION attribute can contain the URL of a CGI program that will process
  the data entered by the user when the user indicates that they wish to submit the
  form.
• Java applets are included in an HTML page using the `<APPLET>` tag. Since applets are capable of communicating through a socket connected to an arbitrary port on the applet’s server of origin, the applet should be downloaded directly from that server. Otherwise the default is to obtain the applet’s code from the server that supplied the HTML page containing the applet. An applet’s code can be obtained from an arbitrary host on the Internet, specified using the `CODEBASE` attribute. For spoofing to function properly in the presence of Java applets, the `CODEBASE` attribute must be added if not present. If a `CODEBASE` attribute is already present, it must be ensured that the `CODEBASE` attribute is an absolute URL.

5.3 Application of Web Spoofing to Distributed Computing

The Secure Internet Programming Group suggest that Web Spoofing allows tampering with the pages returned to the user, by inserting “misleading or offensive material” [141]. We observe that the opportunity to tamper with the pages allows a Java applet to perform part of a distributed computation to be inserted into the page. Tampering with the spoofed pages in this manner and for this purpose has not been previously suggested.

As Web pages being spoofed must have their URLs rewritten to point at the attacking server, it is a simple matter to insert the HTML code to include a Java applet into each page in the course of performing the other modifications to the page. The Java applet can be set to have only a small “window” on the page, which makes it difficult for users to detect its presence on the Web pages that they view.

When the browser encounters a Java applet for the first time during a session, it usually starts Java, displaying a message to this effect in the status line of the browser. It may be possible to conceal this using JavaScript if necessary (although this has not been tested); however, if Java applets become increasingly common and therefore unremarkable, concealment may be deemed unnecessary.

Users may bookmark spoofed pages during the course of their session. If this occurs, the user will rejoin the computation when next that bookmark is accessed.
5.3. Application of Web Spoofing to Distributed Computing

Rather than knowingly (re)visiting a particular site to acquire a copy of an applet, the user unknowingly contacts an attacking server which incorporates the applet into each page supplied to the user.

A site that employed distributed computing with Java applets and Web spoofing could potentially be running applets not only on machines of browser users who visit the site directly, but also on the machines of users who visit a bookmark made after having directly visited the site on some previous occasion. Thus, the "pool" of users who could be contributing to a computation is not limited to those that directly visit the site, as it is with other approaches to covert distributed computing with Java. For example, consider a Web server that receives on average 10,000 hits/day. If the operators of the Web server elect to incorporate an applet to perform distributed computation in each page downloaded from the server, they will steal some CPU time from an average of 10,000 computers each day. However, by using Web spoofing as well, on the second day after starting the spoofing attack and supplying the applet, CPU time is being stolen from the (on average) 10,000 users who knowingly (re)visit the site, and also from users who have visited the site on the previous day and made bookmarks to other sites subsequently visited.

The level of load on the attacking server can be controlled by redirecting if necessary some requests directly to the actual server containing the resource, foregoing the opportunity to perform Web Spoofing, and of stealing computation time from some unsuspecting browser user, but keeping the load on the server at reasonable levels.

An attacker might decide to increase the likelihood of bookmarks referring to spoofed pages by modifying a Web search engine to return answers to queries that incorporate spoofed links, but not require the search engine itself to participate in the spoofing. Of course, if the pages of an unmodified search engine are being spoofed when a search is performed, rewriting of the URLs in the response will take place automatically.

The user's Web browser will display a URL in its location line which exhibits the presence of spoofing when revisiting a bookmark made of a spoofed page; however, we believe that this may be concealed after the page has commenced loading using JavaScript, although again this has not been implemented.
5.3.1 An Implementation

For reasons of simplicity, the Web spoofing attack for the purposes of demonstration was implemented using a CGI program. An off-the-shelf Web server was used to handle HTTP requests.

The applet to demonstrate the performance of a simple key cracking task was of course implemented in Java. The program which kept track of which subproblems had been completed (without finding a solution) and that distributed new subproblems to client applets was also written in Java (the "problem server").

Client applets use threads, one for performing a computation, another for communicating with the server to periodically report the status of their particular computation to the problem server. Periodic reporting to the server guards against the loss of an entire computation should the client applet be terminated before completion of the entire computation; this might occur as a result of the user exiting the Web browser in which the applet is being run, for example.

The thread performing the computation sleeps periodically, to avoid using excessive resources and so unintentionally revealing its presence.

The problem server keeps records using the IP numbers of the computers on which a client applet is running, and will allow only one instance of an applet to run on each computer, to avoid degrading performance too noticeably. A new client applet will be permitted to commence operations if some amount of time has elapsed without a report from the original client applet.

5.4 Countermeasures?

5.4.1 Client-side Precautions

Obviously, disabling Java is an excellent way for a user to ensure that he or she does not participate unwillingly in such an attack, as the Java applet to perform the computation will be unable to run. The disadvantage of this approach is that applets performing services of potential utility to the user will also not be able to run.

The merits of disabling JavaScript are briefly discussed by the Princeton group [141]. This prevents the Web spoofing from being concealed from the clients.
5.4. Countermeasures?

How many clients would take notice of the signs is an interesting question, especially given that in the future clients may have become accustomed to the use of strange URLs such as those produced by a Web spoofing program, as there are several sites providing legitimate services with a Web spoofing-style program.

5.4.2 Server-side Precautions

During the preparation of a demonstration of this approach to covert distributed computing with Java, it was observed that there were some sites whose pages included counters of the number of times that a site had been visited, and links to other CGI programs, all of which failed to produce the expected results when the page containing the counter, or link to CGI program, was being spoofed. Note that a page visit counter is commonly implemented by using a CGI program through an <IMG> HTML tag.

The problem was eventually traced to an improperly set Referer: field in the HTTP request sent by the spoofing program to fetch an HTML page from the server being spoofed. The Referer: field that was originally being sent included a spoofed URL.

The Referer: field of an HTTP request is used to specify the address of the resource, most commonly a Web page, that contained the URL reference to the resource which is the subject of the HTTP request, in cases where the URL reference was obtained from a source that may be referred to by a URL; this field would be empty if the source of the request was the keyboard, for example [142, 143].

The value of the Referer: field can be checked by a CGI program to determine that a request to execute a CGI program comes only from a URL embedded in a specific page, or a set of pages. This prevents easy misuse or abuse of the CGI program by others.

An implementation of distributed computing with Java in the manner described in this chapter would want to keep the amount of data passing through the attacking server as small as possible, to minimise response time to client requests, and so that the number of clients actively fetching pages and performing computations could be maximized. To achieve this, only URLs in pages which are likely to point at an HTML

\[\text{\textsuperscript{1}}\text{"The Anonymizer": see http://www.anonymizer.com/ [141]; the "Zippy Filter": see http://www.metahtml.com/apps/zippy/welcome.mhtml [141]; the "Fool's Tools" have been used to "reshape" HTML: http://las.alfred.edu/~pav/fooltools.html.}\]
5.4. Countermeasures?

document (whose URLs will need to be rewritten, so that the client continues to view spoofed documents) are rewritten to point at the spoofing server — all other URLs, specifically URLs in HTML `<IMG>` tags are modified only so that they fully specify the server and resource path; they are *NOT* spoofed.

It is easy enough for the attacking server to adjust the `Referer:` field to have the value it would normally have, were the page not being spoofed. However, this does not help with the fetching of non-spoofed resources such as images — the attacking server never sees the HTTP request for these resources. So the `Referer:` field will not be set as the spoofed server would expect.

So we propose that an effective countermeasure against a spoofing attack implemented for the purpose of performing a distributed computation is for Web servers to check the `Referer:` field for images and other resources that are expected to be always embedded in some page being served by the Web server for consistency; that is, the `Referer:` field indicates always that the referrer of the document is a page served by the Web server. Usually it should be sufficient to verify that the address of the Web server that served the page which contained the URL for the resource currently being served is the same as the Web server asked to serve the current request. The Web server could refuse to serve a resource if its checks of the `Referer:` field were not satisfied, or display an alternative resource, perhaps attempting to explain possible causes of the problem.

While checking the `Referer:` field and taking action depending on its contents does not prevent an attack from taking place, it does mean that unless all the images contained on a page are also spoofed there will be gaps where a Web server has refused to serve an image because its checks of the `Referer:` field have failed. Given the high graphical content of many Web pages, it is unlikely that a user would wish or be able to persist in their Web travels while the pages were being spoofed. Either they would find a solution or stop using the Web. Spoofing all the images would increase the amount of data processed by the spoofing server, which as a result would greatly limit the number of clients who could be effectively spoofed at the one time.

Unfortunately, at this time there are some inconsistencies in the way in which different Web browsers handle the `Referer:` field. Common Web browsers like Netscape Navigator and Microsoft Internet Explorer appear to provide `Referer:`
fields for HTTP requests for images embedded in Web pages, for example. Other less widely used browsers, such as Apple Computer’s Cyberdog, do not do so.

It should be noted that this countermeasure would be most effective in protecting users if many of the Web servers in existence were to implement this sort of check. A site could, however, implement this countermeasure to help ensure that users of that particular site were likely to detect the presence of Web spoofing.

5.5 Computing for Sale

It is not unusual to find that a Java applet that performs a distributed computation is classed as a “malicious” applet [137, pp. 113–114] [139]. The computation is undesirable because the user is not aware that it is being performed.

On the other hand, it is not difficult to imagine a large group of users donating some of their computer time to help perform a long computation. Examples include efforts to crack instances of DES or RC5 encrypted messages\(^2\), or finding Mersenne Primes\(^3\). Using Java for this sort of purpose avoids many troublesome issues of producing a client program for a variety of different computer platforms; there is, however, currently a heavy speed penalty that must be paid, as Java is not as efficient as a highly-tuned platform-dependent implementation. Improvements in JIT compilers, mentioned earlier, will help to reduce this speed penalty, but will not eliminate it entirely.

We introduce the idea of “Computing for Sale” — that sites which provide some form of service to clients could require that clients allow the running of a Java applet for some fixed period of time as the “price” for accessing the service. An excellent example of a service to which this idea could be applied is that of a Web search engine, or perhaps an online technical reference library or support service. Clients that are unable or unwilling to allow the Java applet to run so that it may perform its computation could be provided with a reduced service. For example, a client of a Web search engine who refused to run the applet could be provided with E-mail results of their query half an hour or so after query submission rather than immediately. This provides an incentive for clients to allow computation applets to perform their tasks.

\(^2\)See http://www.distributed.net/
\(^3\)See http://www.mersenne.org/
It would be possible for a service to deny access or provide only a reduced service to a client whose computation applets consistently fail to report results for some reason, such as being terminated by the client.

In the interests of working in a wide variety of network environments, applets used for this purpose should be able to communicate with the server using methods apart from ordinary socket connections, such as, by using Java’s URL access capabilities, HTTP POST or GET messages [144]. For example, a firewall might prohibit arbitrary socket connections originating behind the firewall but permit HTTP message traffic.

The service provider would be able to sell computation time in much the same way as many providers sell advertising space on their Web pages. Some clients of such services might prefer to choose between an advertising-free service which requires that client assist in performing a computation, and the usual service loaded with advertising, but not requiring the client to assist with the computation by running an applet.

5.6 Conclusion

There are many problems in computer science which can best be solved by the application of brute force. An example is the determination of an unknown cryptographic key, given some ciphertext and corresponding plaintext. Distributed computing offers a way of obtaining the necessary resources, by using a portion of the CPU time of many computers.

Such a project can either be conducted with full knowledge and cooperation of all participants, or covertly. There have been some suggestions that applets written in Java and running in Web browsers might perform covert distributed computations without the knowledge of browser users, but requiring browser users to knowingly visit a particular site.

We observe that Web Spoofing offers a way of not only adding Java applets to perform covert distributed computations to Web pages, but also of increasing the likelihood of past unwitting contributors contributing again when they revisit bookmarks made during a prior spoofed Web browsing session.

Some simple measures which make a successful attack more difficult and less
5.6. Conclusion

likely have been examined. These include disabling Java. We also proposed that servers examine the `Referer:` field of HTTP requests, and refuse to serve the object, or serve some other object explaining the problem, should the `Referer:` field not be consistent with expectations.

We introduced the idea of browser users "paying" for access to services and resources on the Web through the use of their idle computer time for a short period. Service providers could then sell these CPU resources in the same way as advertising is now sold, and may feel it appropriate to offer an advertising-free service for users who "pay" using their computer's idle time.
Part II

Some Applications of Distributed Computing to Combinatorial Search
Chapter 6

Introduction to Sequences and Hadamard Matrices

6.1 Sequences

6.1.1 Introduction

Sequences have many practical and mathematical applications. For example, *Barker sequences*, short sequences of ±1, may be used in radar systems to improve range resolution and accuracy [145]. No Barker sequences longer than 13 elements are known, and it is believed that none exist.

As another example, *m-sequences* or *maximal length binary sequences* are sequences composed of 0 and 1 elements. Such sequences are output from certain configurations of linear feedback shift register (LFSR). m-sequences may also be used in radar systems in place of Barker sequences if a larger sequence than is available with a Barker sequence is required [145].

Another application of m-sequences is in the area of digital watermarking [146, 147, 148]. Digital watermarking is a very active field of research.

A more mathematical application of sequences is to create new weighing matrices or orthogonal designs. In this section, we expand further on the mathematical background presented to the sequence search in Chapter 7. The use of sequences in constructing weighing matrices and orthogonal designs has been studied extensively. The main references for this section are Geramita and Seberry [149] and a review of the construction of weighing matrices and orthogonal designs using two sequences with zero autocorrelation function by Koukouvinos and Seberry [150].
6.1.2 Mathematical Background

Given a set of \( l \) sequences, the sequences \( A_j = \{a_{j1}, a_{j2}, \ldots, a_{jn}\}, j = 1, \ldots, l \) each of length \( n \), the periodic autocorrelation function \( P_A(s) \) is defined as:

\[
P_A(s) = \sum_{j=1}^{l} \sum_{i=1}^{n} a_{ji} a_{j,i+s}, s = 0, 1, \ldots, n - 1
\]

There is also a non-periodic autocorrelation function, \( N_A(s) \), defined as:

\[
N_A(s) = \sum_{j=1}^{l} \sum_{i=1}^{n-s} a_{ji} a_{j,i+s}, s = 0, 1, \ldots, n - 1
\]

Note that \( P_A(s) = N_A(s) + N_A(n - s) \), so that a set of sequences with zero non-periodic autocorrelation function also has zero periodic autocorrelation function.

Here we are interested in sets of two sequences of length \( n \). These will be said to be of type \( (s_1, s_2) \) if the sequences are composed of two variables, say \( x_1 \) and \( x_2 \), and \( \pm x_1 \) occurs a total of \( s_1 \) times and \( \pm x_2 \) occurs a total of \( s_2 \) times. Other elements of the sequences are zero. Such sequences may be used as the first rows of two circulant matrices to obtain an orthogonal design \( OD(2n; s_1, s_2) \). We note that this means that the two sequences have zero (non-)periodic autocorrelation function.

An orthogonal design is defined as follows:

**Definition 1** An orthogonal design of order \( n \) and type \( (s_1, s_2, \ldots, s_u) \), \( s_i \) positive integers, is an \( n \times n \) matrix \( X \) with entries \( \{0, \pm x_1, \ldots, \pm x_u\} \) (the \( x_i \) commuting indeterminates) satisfying:

\[
XX^T = \left( \sum_{i=1}^{u} s_i x_i^2 \right) I_n
\]

This is written as \( OD(n; s_1, s_2, \ldots, s_u) \).

Alternatively, an orthogonal design can be thought of as an \( n \times n \) matrix \( X \) in which each row (column) has \( s_i \) entries of the type \( \pm x_i \), and the distinct rows (columns) are orthogonal under the euclidean inner product.

Two sequences are said to be of type \( (0, \pm 1) \) and weight \( w \) if they have a total of \( w \) non-zero elements \( \pm 1 \), and may be used as the first rows of two circulant matrices to obtain a weighing matrix \( W(2n, w) \). We note that this means that the two sequences
6.1. Sequences

have zero (non-)periodic autocorrelation function. A weighing matrix is defined as follows:

**Definition 2** A weighing matrix \( W = W(n, k) \) is a square matrix with entries 0, ±1 having \( k \) non-zero entries per row and column and inner product of distinct rows zero. \( W \) satisfies \( WW^T = kI_n \), and \( W \) is equivalent to an orthogonal design \( OD(n; k) \).

Weighing matrices have been studied because of their use in improving the accuracy of weighing experiments, as first studied by Hotelling [151].

The maximum number of variables in an orthogonal design is given by the Radon number, defined by \( \rho(n) = 8c + 2^d \) where \( n = 2^a b \), \( b \) odd, and \( a = 4c + d \), \( 0 \leq d < 4 \).

We are interested in orthogonal designs formed from two circulant matrices of odd dimension, and so we are limited to designs of one variable (weighing matrices) and two variables.

To form the orthogonal design from the two circulant matrices, the following theorem is used:

**Theorem 1 (Geramita-Seberry [149])** If there exist two circulant matrices \( A_1, A_2 \) of order \( n \) satisfying:

\[
\sum_{i=1}^{2} A_i A_i^T = fI
\]

and \( f \) is the quadratic form \( \sum_{j=1}^{2} s_j x_j^2 \), then there is an orthogonal design \( OD(2n; s_1, s_2) \). If \( f \) is an integer there exists a \( W(2n, f) \).

The circulant matrices \( A_1, A_2 \) are used as follows:

\[
D = \begin{pmatrix} A_1 & A_2 \\ -A_2^T & A_1^T \end{pmatrix} \quad \text{or} \quad D = \begin{pmatrix} A_1 & A_2 R \\ -A_2 R & A_1 \end{pmatrix}
\]

where \( R \) is the back diagonal \((0, 1)\) matrix.

6.1.3 Asymptotic Existence

**Theorem 2 (Eades [149])** Suppose there is a 2 × 2 rational matrix \( P \) such that \( PP^T = \text{diag}(s_1, s_2) \), where \( s_1 \) and \( s_2 \) are positive integers. Then there is an integer \( N = N(s_1, s_2) \) such that for all \( t \geq N \) there is an \( OD(2t; s_1, s_2) \).
6.2 Hadamard Matrices

6.2.1 Introduction

Hadamard matrices have a long and extensive history of study. In this section, we briefly cover some of the basic theory applying to Hadamard matrices. We will look at a number of constructions for Hadamard matrices. The construction that we use in our distributed search is called Williamson’s method; it is discussed in greater detail in Chapter 8. Many similar constructions exist, some of which will also be mentioned.

We will look briefly at how Hadamard matrices can be used to construct codes to be used in communications. Hadamard matrices have many applications; we consider this one in particular because of its accessibility to any audience.

Williamson’s construction has been used as the basis of a number of exhaustive searches for Hadamard matrices. We describe the results from these prior searches.

The main reference for this section is Seberry and Yamada [152].

6.2.2 Basic Theory

An Hadamard matrix $H$ of order $h$ is a square matrix with entries $\pm1$, such that:

$$HH^T = hI_h$$  \hspace{1cm} (6.4)

That is, the row vectors of $H$ are pairwise orthogonal.

These matrices were first studied by J. J. Sylvester [153].

The rows and columns of $H$ may be permuted without altering the property specified by Equation 6.4. Entries in rows and columns may also be multiplied by $-1$ without altering this property. Two Hadamard matrices are said to be equivalent if one can be obtained from the other by a sequence of operations of these types.

Some basic properties of Hadamard matrices are:

**Lemma 1** Let $H$ be an Hadamard matrix of order $h$. Then the following hold:
6.2. Hadamard Matrices

1. \( HH^T = hI_h \)

2. \(|\det H| = h^{\frac{1}{2}}\)

3. \( HH^T = H^T H \)

4. Every Hadamard matrix is equivalent to an Hadamard matrix that has every element of its first row and column +1. Matrices of this latter form are called normalized.

5. \( h = 1, 2 \) or \( 4n \), \( n \) an integer.

6. If \( H \) is a normalized Hadamard matrix of order \( 4n \), then every row (column) except the first has \( 2n-1 \)'s and \( 2n+1 \)'s in each row (column); further, \( n-1 \)'s in any row (column) overlap with \( n-1 \)'s in each other row (column).

6.2.3 Some constructions for Hadamard Matrices using “multiplication”

Sylvester [153] observed that if \( H \) is an Hadamard matrix, then

\[
\begin{pmatrix}
H & H \\
H & -H
\end{pmatrix}
\]

is also an Hadamard matrix. Using the Hadamard matrix of order 2:

\[
\begin{pmatrix}
1 & 1 \\
1 & -1
\end{pmatrix}
\]

we have:

Lemma 2 (Sylvester [153]) There is an Hadamard matrix of order \( 2^t \) for all integers \( t \).

Definition 3 If \( M = (m_{ij}) \) is a \( m \times p \) matrix and \( N = (n_{ij}) \) is an \( n \times q \) matrix, then the Kronecker product \( M \times N \) is the \( mn \times pq \) matrix given by:

\[
M \times N = \begin{bmatrix}
m_{11}N & m_{12}N & \cdots & m_{1p}N \\
m_{21}N & m_{22}N & \cdots & m_{2p}N \\
\vdots & \vdots & \ddots & \vdots \\
m_{m1}N & m_{m2}N & \cdots & m_{mp}N
\end{bmatrix}
\]
Lemma 3 (Hadamard [154]) Let \( H_1 \) and \( H_2 \) be Hadamard matrices of orders \( h_1 \) and \( h_2 \). Then \( H = H_1 \times H_2 \) is an Hadamard matrix of order \( h_1 h_2 \).

Lemma 4 (The Multiplication Theorem of Agayan-Sarukhanyan [155]) Let \( H_1 \) and \( H_2 \) be Hadamard matrices of orders \( 4h \) and \( 4k \). Then there is an Hadamard matrix of order \( 8hk \).

Theorem 3 (Craigen-Seberry-Zhang [156]) Suppose that there are Hadamard matrices of orders \( 4a, 4b, 4c, 4d \). Then there is an Hadamard matrix of order \( 16abcd \).

Definition 4 An Hadamard matrix \( H \) is said to be regular if the sum of all the elements in each row or column is a constant \( k \). Hence \( HJ = JH = kJ \), where \( J \) is the matrix of all ones.

Theorem 4 (Goethals-Seidel [157]) Suppose that there is an Hadamard matrix of order \( h \). Then there is a regular symmetric Hadamard matrix with constant diagonal of order \( h^2 \).

Definition 5 A matrix \( C \) of order \( 2n \) with elements \( \pm 1, \pm i \) that satisfies \( CC^* = 2nI \) will be called a complex Hadamard matrix.

Theorem 5 (Turyn [158]) Suppose that there is a complex Hadamard matrix of order \( 2n \) and an Hadamard matrix of order \( 4h \). Then there is an Hadamard matrix of order \( 8hn \).

6.2.4 Direct construction theorems for Hadamard Matrices

Many constructions for Hadamard matrices require that matrices be “plugged in” to some other matrix. These constructions are the main subject of this section.

Theorem 6 (Paley [159]) Let \( p \equiv 3 \mod 4 \) be a prime power. Then there is an Hadamard matrix of order \( p + 1 \).

Theorem 7 (Paley [159]) Let \( p \equiv 1 \mod 4 \) be a prime power. Then there is an Hadamard matrix of order \( 2(p + 1) \).
6.2. Hadamard Matrices

\[ \begin{bmatrix} A & B & C & D \\ -B & A & -D & C \\ -C & D & A & -B \\ -D & -C & B & A \end{bmatrix} \]

the right representation of the quaternions;

\[ \begin{bmatrix} A & B & C & D \\ -B & A & D & -C \\ -C & -D & A & B \\ -D & C & -B & A \end{bmatrix} \]

the left representation of the quaternions.

Figure 6.1: The Williamson Array

**Definition 6** X and Y are said to be amicable matrices if

\[ XY^T = YX^T \]

**Definition 7** Four circulant, symmetric ±1 matrices A, B, C, D of order w that satisfy

\[ AA^T + BB^T + CC^T + DD^T = 4wI_w \]

will be called Williamson matrices. Four ±1 matrices A, B, C, D of order w that satisfy both

\[ XY^T = YX^T \text{ for } X, Y \in \{A, B, C, D\} \]

(that is, A, B, C, D are pairwise amicable), and

\[ AA^T + BB^T + CC^T + DD^T = 4wI_w \] (6.5)

will be called Williamson-type matrices.

The most common structure matrices are “plugged into” is the orthogonal design. Orthogonal designs are defined in Definition 1.

An orthogonal design with no zeros and in which each entry is replaced with a +1 or −1 is an Hadamard matrix.

Some orthogonal designs that are of interest are:

1. the Williamson array: the OD(4, 1, 1, 1). The Williamson array is shown in Figure 6.1;
2. an OD(8;1,1,1,1,1,1,1,1) is known;

3. the Baumert-Hall array: the OD(12;3,3,3,3);

4. the Plotkin array: the OD(24;3,3,3,3,3,3,3,3);

5. the Welch array: the OD(20;5,5,5,5);

6. the Ono-Sawade-Yamamoto array: the OD(36;9,9,9,9);

7. the Goethals-Seidel array [160]. The array is shown in Figure 6.2, where \( A, B, C, D \) are circulant matrices satisfying Equation 6.5 and \( R \) is the back diagonal \((0,1)\) matrix.

It remains only to define conditions on matrices that are “plugged into” these orthogonal designs to form an Hadamard matrices:

**Definition 8** Suitable matrices of order \( w \) for an OD\((a_1,a_2,\ldots,a_u)\) are \( u \) pairwise amicable matrices (that is, that pairwise satisfy Definition 6), \( A_i, i = 1,\ldots,u \), that have entries +1 or −1 and that satisfy:

\[
\sum_{i=1}^{u} s_i A_i A_i^T = (\sum_{i} s_i) w I_w
\]

(6.6)

Suitable matrices are used in the following theorem:

**Theorem 8 (Geramita-Seberry)** Suppose that there exists an OD\((\sum a_i; s_1, s_2,\ldots, s_u)\) and \( u \) suitable matrices of order \( m \). Then there is an Hadamard matrix of order \((\sum a_i)m\).

### 6.2.5 Hadamard Matrices and Block Codes

Hadamard matrices have many applications [161]. One use is the construction of block codes.
Consider, for example, the problem of transmitting a text message from one place to another. Each letter will be encoded for transmission by representing it using its 8-bit ASCII binary value. For example, A is represented by 01000001.

It is not uncommon for errors to occur during transmission; most real-world channels are noisy. Consider what happens if a 1-bit error occurs in the transmission of 'A': 01000001 is sent but 01001001 is received, for example. The received codeword corresponds to the character 'T'.

So the choice of 8-bit ASCII binary values to encode a text message is not a particularly good one: all codewords are being used, and a single-bit error is sufficient to change one character being transmitted into another. It is not even possible to detect that this has occurred.

However, we can create codes which give some degree of error-detecting and error-correcting capability. The Hamming distance between two codewords is the number of places in which the codewords disagree. A code is a “distance $d$ code” when the minimum Hamming distance between any pair of codewords is at least $d$. A distance $d$ code can detect $(d-1)$ errors in the transmission of a codeword, and can correct $\frac{1}{2}(d-1)$ errors in the transmission of a codeword.

**Definition 9** An $(n,k)$ linear code over the field $GF(q)$ consists of $q^k$ vectors (called codewords) of length $n$ with components from $GF(q)$ such that:

1. the vector sum of two codewords is a codeword;

2. the multiplication of any codeword by a scalar which is any element of $GF(q)$ yields a codeword.

Hadamard matrices may be easily used to form non-linear binary codes.

**Definition 10** An $(n,M,d)$ code is a set of $M$ codewords, all of length $n$, composed with symbols from $GF(q)$ symbols and minimum Hamming distance $d$. An $(n,M,d)$ code is called optimal if $M$ is as large as possible for the given $n$ and $d$.

We consider binary codes here. In this case, $q = 2$, and the symbols used in constructing codewords are 0 and 1.

**Theorem 9** The existence of an Hadamard matrix of order $4t$ implies the existence of the following optimal codes:
6.2. Hadamard Matrices

- $(4t, 8t, 2t)$;
- $(4t - 1, 4t, 2t)$;
- $(4t - 1, 8t, 2t - 1)$; and
- $(4t - 2, 2t, 2t)$.

Codes constructed using Hadamard matrices have seen practical applications: a $(32, 64, 16)$ code was used in the Mariner telemetry system in 1969.

For more details on constructing binary codes using Hadamard matrices, consult [161].

6.2.6 Computer Searches for Hadamard Matrices of Williamson Type

In many cases complete searches have been conducted for Hadamard matrices of Williamson type. Searches have also been conducted for special classes of Williamson type Hadamard matrices. Furthermore, an infinite class of such matrices is known and will also be discussed briefly.

- Baumert and Hall [162] report results of a complete search for orders $4t$, $t$ odd and $3 \leq t \leq 23$. Some incomplete results for higher orders are also given.

- Sawade [163] reports results of a complete search for orders $4t$, $t = 25, 27$. The results for $t = 25$ were later demonstrated to be incomplete by Dokovic [164].

- Dokovic [165] reports results of a complete search for orders $4t$, $t = 29, 31$. Only a single non-equivalent solution was found for $t = 29$ and is equivalent to an earlier result due to Baumert [166].

- Koukouvinos and Kounias [167] report results of a complete search for order $4t$, $t = 33$. These results were later demonstrated to be incomplete by Dokovic [168].

- Dokovic [168] reports results of a complete search for orders $4t$, $t = 33, 35, 39$. 

6.2. Hadamard Matrices

- Dokovic [164] reports results of a complete search for orders $4t$, $t = 25, 37$. This extends results obtained by Sawade [163] for $t = 25$ and, for $t = 37$, by Williamson [169] and later Yamada [170] for a special class of matrices.

An infinite family of Hadamard matrices of Williamson type has been proved to exist under certain conditions [171, 172]:

**Theorem 10** If $q$ is a prime power, $q \equiv 1 \pmod{4}$, $q + 1 = 2t$, then there exists a Williamson matrix of order $4t$; we have $C = D$, and $A$ and $B$ differ only on the main diagonal.

This theorem gives examples of Hadamard matrices of Williamson type for orders $4t$, $t = 31, 37, 45, 49, 51, 55, \ldots$, for example.

Yamada [170] has searched for Hadamard matrices of Williamson type, with certain restrictions. These matrices are referred to as Williamson type $j$ matrices. The Williamson equation for such matrices, of order $4n$ is:

$$4n = \left(1 - 2 \sum_{s \in A} c_s \omega_s \right)^2 + \left(1 - 2 \sum_{s \in A} c_s \omega_{sj} \right)^2 + \left(1 - 2 \sum_{s \in B} d_s \omega_s \right)^2 + \left(1 - 2 \sum_{s \in B} d_s \omega_{sj} \right)^2 \quad (6.7)$$

where $c_s, d_s = \pm 1$, $\omega_s = \omega^s + \omega^{-s}$, $\omega^n = 1$, $j^2 \equiv 1 \pmod{n}$, $A, B, jA, jB$ is a partition of \{1, 2, $\ldots$, $\frac{n-1}{2}$\}. Such a $j$ exists if and only if all prime divisors of $n$ are $\equiv 1 \pmod{4}$. This led to some new results for $n = 29, 37, 41$. 

Chapter 7

When the necessary conditions are not sufficient: sequences with zero autocorrelation function

The material appearing in this chapter has been published as:


7.1 Introduction

Recently K. T. Arasu (personal communication) and Yoseph Strassler, in his PhD thesis, *The Classification of Circulant Weighing Matrices of Weight 9*, Bar-Ilan University, Ramat-Gan, 1997, have intensively studied circulant weighing matrices, or single sequences, with weight 9. They show many cases are non-existent. Here we give details of a search for two sequences with zero periodic autocorrelation and types (1,9), (1,16) and (4,9). We find some new cases but also many cases where the known necessary conditions are not sufficient.

We instance a number of occasions when the known necessary conditions are not sufficient for the existence of weighing matrices and orthogonal designs constructed using sequences with zero autocorrelation function leading to intriguing new questions.

A weighing matrix $W = W(n,k)$ is a square matrix with entries 0, $\pm 1$ having $k$ non-zero entries per row and column and inner product of distinct rows zero. Hence $W$ satisfies $WW^T = kI_n$, and $W$ is equivalent to an orthogonal design $OD(n;k)$ (see
Geramita and Seberry [149] for more details). The number $k$ is called the weight of $W$.

Weighing matrices were first studied by Hotelling [151] because of their use in weighing experiments.

Given a set of $\ell$ sequences, the sequences $A_j = \{a_{j1}, a_{j2}, \ldots, a_{jn}\}, j = 1, \ldots, \ell$, of length $n$ the periodic autocorrelation function, $PAF, P_A(s)$, is defined, reducing $i+s$ modulo $n$, as

$$P_A(s) = \sum_{j=1}^{\ell} \sum_{i=1}^{n} a_{ji} a_{j, i+s}, \quad s = 0, 1, \ldots, n-1. \quad (7.1)$$

Two sequences, of length $n$, will be said to be of type $(s, t)$ if the sequences are composed of two variables, say $a$ and $b$, and $a$ and $-a$ occur a total of $s$ times and $b$ and $-b$ occur a total of $t$ times. Such sequences may be used as the first rows of two circulant matrices to obtain an orthogonal design, $OD(2n; s, t)$. The sequences are said to be of type $(0, \pm 1)$ and weight $w$ if they have a total of $w$ non-zero elements and may be used as the first rows of two circulant matrices to obtain a $W(2n, w)$.

We note, using the languages of sequences, the following conditions from [149]. In addition we note there is no square orthogonal matrix of size 18 which will have one element once in each row and column and the other 16 times.

**Theorem 11 (Necessary Conditions)** The necessary conditions for the existence of a single sequence of length $n$, $n$ odd, type $(0, \pm 1)$ and weight $k$ are that $k$ is a perfect square and $(n-k)^2 - (n-k) \geq n-1$.

The following conditions are necessary for the existence of two sequences type $(s, t)$ and length $n$ with $n$ odd:

i) $s$, $t$ and $s+t$ must each be the sum of two squares,

ii) $s+t \leq 2n-1,

iii) there exists a $2 \times 2$ integer matrix $P$ (called the sum-fill matrix), with all entries of modulus $\leq n$ which satisfies $PP^T = diag(s, t)$.

The necessary conditions for the existence of four sequences of types $(s_1, s_2), (s_1, s_2, s_3)$ or $(s_1, s_2, s_3, s_4)$ are given in [149, 173].

In this note we concentrate on sequences of types $(1, 9)$, $(1, 16)$ and $(4, 9)$ which satisfy all known necessary conditions. We undertook a computer search which was
unable to find sequences of type (4,9) with a zero periodic autocorrelation function except for lengths $19m$ and $21m$, $m \geq 1$. For types (1,9) and (1,16) we were able to find several new sequences. This leads to the intriguing question of what other necessary conditions must hold for these sequences to exist.

The large number of possibilities to be examined in the course of an exhaustive search for sequences with zero autocorrelation function suggests that a solution based around distributed computing may be useful. Fortunately, in this case, an efficient algorithm, described in Section 7.4.1, was identified which enabled the original goals of the search to be met and surpassed, without explicitly constructing an algorithm based around distributed computing, a process which can be both difficult and time-consuming. The algorithm employed was distributable to a very limited degree: the space to be searched could be broken up easily into a number of parts of varying size, each of which could be processed by different computer. However, this feature was often of limited use due to the very uneven size of the subspaces of the search and the amount of work involved in processing each one.

The implementation of the program used to perform the search can be found in Appendix B. It is written in the language C and is targeted at compilers running on UNIX and Macintosh systems. If desired, it should be easily possible to port to another system. Some brief instructions for use may be found as a comment placed at the head of the program.

## 7.2 Preliminary Results

We make extensive use of the book of Geramita and Seberry [149]. For convenience, and since [149] is out of print, we restate some of the results in the language of sequences for reference. The next useful lemma is well known.

**Lemma 5** If there exists a sequence of length $n$, type $(0, \pm 1)$ and weight $k$ then there exists a sequence of length $pn$, type $(0, \pm 1)$ and weight $k$ for all $p \geq 1$.

**Theorem 12** (Circulant Kronecker Product) If there exists a sequence of length $n$, type $(0, \pm 1)$ and weight $k$, and another sequence of length $m$, type $(0, \pm 1)$ and weight $\ell$ where $\gcd(n,m) = 1$ then there exists a sequence of length $mn$, type $(0, \pm 1)$ and
7.3. Weighing matrices of weight 9

weight $kl$. 

**Theorem 13** If there exist two sequences of length $n$ and type $(s,t)$ then there exist two sequences of length $pn$ and type $(s,t)$ for all integers $p \geq 1$.

**Theorem 14** Suppose $q$ is a prime power and $q^2 + q + 1$ is a prime. Then there exists a single sequence of length $q^2 + q + 1$, type $(0, \pm 1)$ and weight $q^2$. And there exist two sequences of length $q^2 + q + 1$, type $(0, \pm 1)$ and weight $(q + 1)^2$.

If, in addition, $q^2 + q + 1 \equiv 3 \pmod{4}$ is prime, then there exist two sequences of length $q^2 + q + 1$ and type $(1, (q + 1)^2)$.

**Remark 1** We note that for $q = 2$ this means there are two sequences of length 7 and type $(1,9)$; and for $q = 3$ there are two sequences of length 13 and type $(1,16)$. There also exists a sequence of length 21 and type $(1,16)$, using the $W(21,16)$ from Theorem 14 plus $W(21,1)$. 

The first part of Theorem 14 allows us to construct a sequence of length 7, type $(0, \pm 1)$ and weight 4; a sequence of length 13, type $(0, \pm 1)$ and weight 9; and a sequence of length 21, type $(0, \pm 1)$ and weight 16. We use Lemma 5 to make a sequence of length $91m$, type $(0, \pm 1)$ and weight 4 and a sequence of length $91m$, type $(0, \pm 1)$ and weight 9. This ensures there are two sequences of length $91m$ and type $(4,9)$ for all $m \geq 1$.

Two sequences of length 11 and type $(1,16)$ are known from [149].

**7.3 Weighing matrices of weight 9**

Strassler [174] has completed the classification of single sequences of length $n$, type $(0, \pm 1)$ and weight 9.

When we consider two sequences of length $n$, type $(0, \pm 1)$ and weight 9 we find they are known for $n = 5, 7, 11, 13, 17, 19, 23$, and multiples of these numbers by $p$, $p \geq 1$.

In a number of cases the only weighing matrix known of weight 9 is described by writing out its elements as in the $W(15,9)$, $W(17,9)$ and $W(18,9)$ (see Koukouvinos and Seberry [150] for details).
7.4 Sequences with Zero Autocorrelation

The sequences of lengths $n$ and types $(1,9)$ and $(4,9)$ have proved even more elusive. The only known sequences for $(4,9)$ are of length $91m$ (see Remark 1). We have found sequences for $(4,9)$ that are of length $19m$ and $21m$.

For order 14 the following matrix, quoted from [149, p331], has orthogonal rows and is of type $(4,9)$ however no such other matrix is known for orders $m \equiv 2 \pmod{4}$, $m \leq 28$.

\[
\begin{array}{cccccccccccccccc}
0 & x & y & y & y & x & x & x & y & y & y & y & y & y & y & y \\
x & 0 & x & x & x & -y & -y & -y & y & y & y & -y & -y & -y & -y & -y \\
y & x & 0 & -y & -y & x & x & -x & y & y & y & y & -y & -y & y & y \\
y & x & -y & 0 & -y & x & x & -x & y & y & y & y & y & -y & y & y \\
y & x & -y & -y & 0 & -x & -x & x & y & y & y & y & y & y & y & y \\
x & -y & x & -x & -x & 0 & y & y & -y & y & y & y & -y & -y & -y & -y \\
x & -y & -x & x & -x & y & 0 & y & y & -y & y & -y & y & -y & y & -y \\
x & -y & -x & -x & x & y & y & 0 & y & y & -y & y & y & y & -y & y \\
y & y & y & y & y & y & y & y & 0 & -x & -x & -y & -x & -x & -y & -x & -x & -y & -x \\
y & y & y & y & y & y & y & y & 0 & -x & -x & -y & -x & -x & -y & -x & -x & -y & -x \\
y & y & y & y & y & y & y & y & 0 & -x & -x & -y & -x & -x & -y & -x & -x & -y & -x \\
y & y & y & y & y & y & y & y & 0 & -x & -x & -y & -x & -x & -y & -x & -x & -y & -x \\
y & -y & y & y & y & y & y & y & 0 & x & x & x & x & x & x & x & x & x & x & x \\
y & -y & y & y & y & y & y & y & 0 & x & x & x & x & x & x & x & x & x & x & x \\
y & -y & y & y & y & y & y & y & 0 & x & x & x & x & x & x & x & x & x & x & x \\
y & -y & y & y & y & y & y & y & 0 & x & x & x & x & x & x & x & x & x & x & x \\
y & -y & y & y & y & y & y & y & 0 & x & x & x & x & x & x & x & x & x & x & x \\
y & -y & y & y & y & y & y & y & 0 & x & x & x & x & x & x & x & x & x & x & x \\
\end{array}
\]

This note details our search for sequences of these types.

7.4.1 Search Method

It is known that sequences $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$ of length $n$ and type $(s,t)$, a variable $a$ and $-a$ occurring $s$ times, and variable $b$ and $-b$ occurring $t$ times, must satisfy the condition:

\[
\left( \sum_{i=1}^{n} a_i \right)^2 + \left( \sum_{i=1}^{n} b_i \right)^2 = sa^2 + tb^2
\]  

(7.2)
We use Equation 7.2 to determine in what quantities each of variables \( a \), \(-a\) and \( b \), \(-b\) can occur in each sequence: this helps limit the size of the space which must be searched for sequences with zero autocorrelation function.

Equation 7.2 is equivalent to:

\[
sa^2 + tb^2 = (N_{A,a}a + N_{A,b}b)^2 + (N_{B,a}a + N_{B,b}b)^2
\]

\[
= (N_{A,a}^2 + N_{B,a}^2)a^2 + (N_{A,b}^2 + N_{B,b}^2)b^2 + 2(N_{A,a}N_{A,b} + N_{B,a}N_{B,b})ab
\]

from which we obtain the following:

\[
N_{A,a}N_{A,b} + N_{B,a}N_{B,b} = 0 \quad (7.3)
\]

\[
N_{A,a}^2 + N_{B,a}^2 = s \quad (7.4)
\]

\[
N_{A,b}^2 + N_{B,b}^2 = t \quad (7.5)
\]

Note that \( N_{X,v} = (\text{count of variable } v) - (\text{count of variable } -v) \) in sequence \( X \).

Consider the case for sequences of type \((4,9)\); the case for \((1,9)\) is similar. Then Equations 7.3, 7.4, 7.5 have solutions:

\[
N_{A,a} = \pm 2 \quad N_{A,b} = 0 \quad N_{B,a} = 0 \quad N_{B,b} = \pm 3
\]

\[
N_{A,a} = 0 \quad N_{A,b} = \pm 3 \quad N_{B,a} = \pm 2 \quad N_{B,b} = 0
\]

We need consider only one of these two possible solutions; the other corresponds to exchanging sequences \( A \) and \( B \).

Each solution has four cases, of which we need to consider only one. The other three cases correspond to changing the sign of each variable in one or both sequences, which does not affect the result of evaluating Equation 7.1.

This yields a table (Table 7.1) of counts of variables present in sequences \( A \) and \( B \) of type \((4,9)\). A similar table (Table 7.2) can be generated for sequences of type \((1,9)\).

We also observe that the result of evaluating Equation 7.1 is not affected by rotating a sequence; we are able to eliminate some rotations of a sequence from the search space by fixing the first element of each sequence. We choose the variable that is to occur in a sequence the fewest number of times as the first (fixed) variable in the sequence; this gives the greatest reduction in the search space.

Note also that the result of evaluating Equation 7.1 is the same for the reflection of a sequence; however, we have only been successful in eliminating a small proportion
7.4. Sequences with Zero Autocorrelation

Table 7.1: Variable counts for sequences A and B of type (4,9).

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>-a</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 7.2: Variable counts for sequences A and B of type (1,9).

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>-a</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

of these from the search space. No reflections have been eliminated from the search space sizes tabulated in Table 7.3.

We can speed up the process of evaluating Equation 7.1 for each pair of possible sequences if we count the number of possible sequences for A and for B given the variable counts for the particular \( s \) and \( t \), evaluate Equation 7.1 for whichever has fewer possible sequences, and store these results in memory\(^1\). Equation 7.1 can then be fully evaluated for both sequences by evaluating it for each possibility of the sequence with the largest number of possibilities together with the results stored in memory.

Table 7.3 records the total size of the search space, amount of the space that was searched before finding a pair of sequences with zero autocorrelation function, and the time taken, for different \((s,t)\) and various \(n\). Results were obtained using three different computers.

\(^1\)Although in practice so much memory may be required for large \(n\) for some \((s,t)\) that this step must be performed in parts. It should also be noted that considerable improvements in runtimes could be obtained by limiting the amount of memory used by a single part to a size that could be accommodated by cache memory.
### Table 7.3: Time taken to search for sequences of types (1,9), (4,9) and (1,16) for various \( n \), multiple computers.

<table>
<thead>
<tr>
<th>Type</th>
<th>( n )</th>
<th>Tot. Sequences (billions)</th>
<th>Searched (billions)</th>
<th>Time (minutes)</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,9)</td>
<td>27</td>
<td>3.78</td>
<td>All</td>
<td>14.5</td>
<td>Ultra-5</td>
</tr>
<tr>
<td>(1,9)</td>
<td>27</td>
<td>3.78</td>
<td>All</td>
<td>21.5</td>
<td>Ultra-2</td>
</tr>
<tr>
<td>(1,9)</td>
<td>29</td>
<td>7.16</td>
<td>All</td>
<td>27</td>
<td>Ultra-5</td>
</tr>
<tr>
<td>(1,9)</td>
<td>29</td>
<td>7.16</td>
<td>All</td>
<td>42.5</td>
<td>Ultra-2</td>
</tr>
<tr>
<td>(1,9)</td>
<td>31</td>
<td>12.94</td>
<td>All</td>
<td>55</td>
<td>Ultra-5</td>
</tr>
<tr>
<td>(1,9)</td>
<td>31</td>
<td>12.94</td>
<td>All</td>
<td>80</td>
<td>Ultra-2</td>
</tr>
<tr>
<td>(1,9)</td>
<td>31</td>
<td>12.94</td>
<td>All</td>
<td>50</td>
<td>PM 7300/200</td>
</tr>
<tr>
<td>(1,9)</td>
<td>33</td>
<td>22.44</td>
<td>All</td>
<td>99</td>
<td>Ultra-5</td>
</tr>
<tr>
<td>(1,9)</td>
<td>33</td>
<td>22.44</td>
<td>All</td>
<td>158</td>
<td>Ultra-2</td>
</tr>
<tr>
<td>(1,9)</td>
<td>33</td>
<td>22.44</td>
<td>All</td>
<td>116</td>
<td>PM 7300/200</td>
</tr>
<tr>
<td>(1,9)</td>
<td>37</td>
<td>60.93</td>
<td>All</td>
<td>275</td>
<td>Ultra-5</td>
</tr>
<tr>
<td>(1,9)</td>
<td>37</td>
<td>60.93</td>
<td>All</td>
<td>416</td>
<td>Ultra-2</td>
</tr>
<tr>
<td>(1,9)</td>
<td>41</td>
<td>148.07</td>
<td>All</td>
<td>767</td>
<td>Ultra-5</td>
</tr>
<tr>
<td>(1,9)</td>
<td>41</td>
<td>148.07</td>
<td>All</td>
<td>1102</td>
<td>Ultra-2</td>
</tr>
<tr>
<td>(1,9)</td>
<td>43</td>
<td>223.01</td>
<td>All</td>
<td>745</td>
<td>R10000</td>
</tr>
<tr>
<td>(1,9)</td>
<td>47</td>
<td>477.31</td>
<td>All</td>
<td>1684</td>
<td>R10000</td>
</tr>
<tr>
<td>(1,9)</td>
<td>47</td>
<td>477.31</td>
<td>All</td>
<td>2550</td>
<td>Ultra-5</td>
</tr>
<tr>
<td>(1,9)</td>
<td>51</td>
<td>956.18</td>
<td>All</td>
<td>3613</td>
<td>R10000</td>
</tr>
<tr>
<td>(1,9)</td>
<td>53</td>
<td>1324.68</td>
<td>All</td>
<td>5132</td>
<td>R10000</td>
</tr>
<tr>
<td>(1,16)</td>
<td>15</td>
<td>10.28</td>
<td>0.68</td>
<td>&lt;1</td>
<td>PM 7300/200</td>
</tr>
<tr>
<td>(1,16)</td>
<td>17</td>
<td>155.83</td>
<td>0.12</td>
<td>&lt;1</td>
<td>PM 7300/200</td>
</tr>
<tr>
<td>(1,16)</td>
<td>19</td>
<td>1538.97</td>
<td>2.39</td>
<td>5</td>
<td>PM 7300/200</td>
</tr>
<tr>
<td>(1,16)</td>
<td>23</td>
<td>63929.02</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(4,9)</td>
<td>15</td>
<td>9.68</td>
<td>All</td>
<td>17</td>
<td>Ultra-5</td>
</tr>
<tr>
<td>(4,9)</td>
<td>15</td>
<td>9.68</td>
<td>All</td>
<td>23.5</td>
<td>Ultra-2</td>
</tr>
<tr>
<td>(4,9)</td>
<td>17</td>
<td>57.96</td>
<td>All</td>
<td>106</td>
<td>Ultra-5</td>
</tr>
<tr>
<td>(4,9)</td>
<td>17</td>
<td>57.96</td>
<td>All</td>
<td>151</td>
<td>Ultra-2</td>
</tr>
<tr>
<td>(4,9)</td>
<td>17</td>
<td>57.96</td>
<td>All</td>
<td>97</td>
<td>PM 7300/200</td>
</tr>
<tr>
<td>(4,9)</td>
<td>19</td>
<td>269.38</td>
<td>71.71</td>
<td>157</td>
<td>Ultra-5</td>
</tr>
<tr>
<td>(4,9)</td>
<td>19</td>
<td>269.38</td>
<td>71.73</td>
<td>245</td>
<td>Ultra-2</td>
</tr>
<tr>
<td>(4,9)</td>
<td>19</td>
<td>269.38</td>
<td>75.02</td>
<td>194</td>
<td>PM 7300/200</td>
</tr>
<tr>
<td>(4,9)</td>
<td>21</td>
<td>1035.27</td>
<td>834.60</td>
<td>1645</td>
<td>Ultra-5</td>
</tr>
<tr>
<td>(4,9)</td>
<td>21</td>
<td>1035.27</td>
<td>833.36</td>
<td>2222</td>
<td>Ultra-2</td>
</tr>
<tr>
<td>(4,9)</td>
<td>23</td>
<td>3431.72</td>
<td>All</td>
<td>9552</td>
<td>Ultra-5</td>
</tr>
<tr>
<td>(4,9)</td>
<td>25</td>
<td>10104.85</td>
<td>All</td>
<td>14230</td>
<td>R10000</td>
</tr>
</tbody>
</table>
7.5. When necessary conditions are not sufficient

<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
<th>Zero PAF</th>
<th>New?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,9)</td>
<td>7</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>(4,9)</td>
<td>7</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>(1,9)</td>
<td>9</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>(1,16)</td>
<td>9</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>(4,9)</td>
<td>9</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>(1,9)</td>
<td>10</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>(4,9)</td>
<td>10</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>(1,9)</td>
<td>11</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>(1,16)</td>
<td>11</td>
<td>Yes</td>
<td>✓</td>
</tr>
<tr>
<td>(4,9)</td>
<td>11</td>
<td>No</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>13</td>
<td>Yes</td>
<td>✓</td>
</tr>
<tr>
<td>(4,9)</td>
<td>13</td>
<td>No</td>
<td>✓</td>
</tr>
<tr>
<td>(1,16)</td>
<td>13</td>
<td>Yes</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>15</td>
<td>Yes</td>
<td>✓</td>
</tr>
<tr>
<td>(1,16)</td>
<td>15</td>
<td>Yes</td>
<td>✓</td>
</tr>
<tr>
<td>(4,9)</td>
<td>15</td>
<td>No</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>17</td>
<td>No</td>
<td>✓</td>
</tr>
<tr>
<td>(1,16)</td>
<td>17</td>
<td>Yes</td>
<td>✓</td>
</tr>
<tr>
<td>(4,9)</td>
<td>17</td>
<td>No</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>19</td>
<td>Yes</td>
<td>✓</td>
</tr>
<tr>
<td>(1,16)</td>
<td>19</td>
<td>Yes</td>
<td>✓</td>
</tr>
<tr>
<td>(4,9)</td>
<td>19</td>
<td>Yes</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>21</td>
<td>Yes</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
<th>Zero PAF</th>
<th>New?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,16)</td>
<td>21</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>(4,9)</td>
<td>21</td>
<td>Yes</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>23</td>
<td>Yes</td>
<td>✓</td>
</tr>
<tr>
<td>(4,9)</td>
<td>23</td>
<td>No</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>25</td>
<td>Yes</td>
<td>✓</td>
</tr>
<tr>
<td>(4,9)</td>
<td>25</td>
<td>No</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>27</td>
<td>No</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>29</td>
<td>No</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>31</td>
<td>No</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>33</td>
<td>No</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>35</td>
<td>Yes</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>37</td>
<td>No</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>39</td>
<td>Yes</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>41</td>
<td>No</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>43</td>
<td>No</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>45</td>
<td>Yes</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>47</td>
<td>No</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>49</td>
<td>Yes</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>51</td>
<td>No</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>53</td>
<td>No</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>55</td>
<td>No</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>57</td>
<td>Yes</td>
<td>✓</td>
</tr>
<tr>
<td>(1,9)</td>
<td>59</td>
<td>No</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 7.4: The results of an exhaustive search for the existence of two sequences of types (1,9), (1,16) and (4,9).

7.4.2 Results

A summary of our results is given in Tables 7.4 and 7.5. Table 7.6 is derived from [150].

The results we now give in Table 7.4 marked ✓ are previously unpublished.

7.5 When necessary conditions are not sufficient

As we have seen in this paper, and from [173, 175, 176], the known necessary conditions are not sufficient for the existence of
7.5. When necessary conditions are not sufficient

Table 7.5: Some new sequences.

<table>
<thead>
<tr>
<th>Len.</th>
<th>Type</th>
<th>Sequences with zero autocorrelation function</th>
<th>Zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>(1,9)</td>
<td>b-b a b-b 0 0;</td>
<td>PAF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b-b b-b b-b 0 0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>(1,9)</td>
<td>a b b-b b-b b-b 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 b b 0 b 0 0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>(1,9)</td>
<td>a 0 b 0 0 0 0 b 0;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b-b b b b b b 0 b 0 0</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>(1,16)</td>
<td>a b-b 0 0 b 0-0 b 0 b-b</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b b b b b b b b-b b 0 0</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>(1,16)</td>
<td>a b-b 0 0 b 0-0 b 0 b-b</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b b b b b b b b-b b 0 0</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>(1,9)</td>
<td>0 0 0 0 b 0 0 a 0 b-b 0 0 b 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b b b-b 0 0 b 0 0 b 0 0</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>(1,16)</td>
<td>a b b 0 0 0 0 0 0 0 0 0 0 0-b-b</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b b b-b b-b b-b b-b b b 0 0</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>(1,16)</td>
<td>a b b 0 0 0 0 0 0 0 0 0 0 0-b-b</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b b b b b b b b-b b-b 0 0</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>(1,9)</td>
<td>0 0 0 0 0 0 0 0 b a-b 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b b b-b b-b b-b 0-b 0 b-b 0-b 0</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>(1,16)</td>
<td>a b 0 0 b 0 0 0 0 0 0 0 0 0 0-b-b</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b b 0 0-b b b b-b b-b 0 b 0 0</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>(4,9)</td>
<td>a a 0 0 b-b-b 0 0 0 0 0 0 b 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>a-a b 0 b 0 0 0 0 0 b 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>(1,9)</td>
<td>b 0 0 b 0 0 0 0 a a 0 0 0 0-b-b 0 0 0-b 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 b 0 0 0 0 0 b 0 0 0 0 0-b 0 0 0-b 0</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>(4,9)</td>
<td>a b b-b 0-b 0 a 0 0 0 0-b-b 0 0 0 b-b 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>a 0 0 0 0 0-a b 0 0 0 0-b-b 0 0 0-b 0</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>(4,9)</td>
<td>a b 0 0 b 0 0 a 0 0-b 0 0-b 0 0 b-b 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>a 0 b 0 0 0-a b 0 0 0 0-a 0 0 0 0 0-b-b</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>(1,9)</td>
<td>b 0 0 0 0 0 0 0 0 0 0 0 a a 0 0 0 0 0 0 0 0-b-b</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b-b b 0 0 b 0 0-b b-b 0 0 0 0 0 0 0 0 0 0 0 0-b-b</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>(1,9)</td>
<td>0 b 0 0 0 0 0 0 0 0 0 0 0-a a 0 0 0 0 0 0 0 0 0 0 0 0-b-b</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b b b-b b-b 0 0 0 0 0 0 0-b-b 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.6: The asymptotic existence of $OD(2n;1,9)$, $OD(2n;1,16)$ and $OD(2n;4,9)$.

<table>
<thead>
<tr>
<th>Type</th>
<th>2n</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,9)</td>
<td>12, 14, 16, (18), 20, (22), 24...</td>
</tr>
<tr>
<td>(1,16)</td>
<td>20...</td>
</tr>
<tr>
<td>(4,9)</td>
<td>14, 16, (18), 20, (22), 24, (26), 28...</td>
</tr>
</tbody>
</table>
• four sequences of lengths 5 and type (3, 7, 8);

• four sequences of lengths 5 and types (1, 3, 6, 8), (1, 4, 4, 9), or (2, 2, 5, 5);

• two sequences of lengths 7, 9, 10, 11, 13, 15, 17, 23 or 25 and type (4, 9);

• four sequences of lengths 7 and type (1, 5, 20);

• four sequences of lengths 7 and types (1, 4, 9, 9), (1, 8, 8, 9), (2, 8, 9, 9), (3, 6, 8, 9), (4, 4, 4, 9) or (4, 4, 9, 9);

• two sequences of lengths 9, 11, 17, 27, 29, 31, 33, 37, 41, 43, 47, 51, 53, 55 or 59 and type (1, 9);

• two sequences of length 9, type (0, ±1) and weight 9;

• four sequences of lengths 9 and types (1, 2, 8, 25), (1, 4, 4, 25) or (2, 3, 4, 24);

• four sequences of lengths 11 and types (5, 38), (6, 37), (8, 35), (10, 33), (12, 31), (13, 30), (14, 29), (15, 28), (16, 27), (19, 24), (20, 23), or (21, 22).

• four sequences of lengths 11 and types (1, 5, 38), (1, 6, 37), (1, 8, 35), (1, 10, 33), (1, 12, 31), (1, 13, 30), (1, 14, 29), (1, 15, 28), (1, 16, 27), (1, 19, 24), (1, 20, 23) or (1, 21, 22).

• single sequence of length 15 or 17, type (0, ±1) and weight 9;

We note there is an (4, 4, 9, 9) constructed of four circulant sequences of lengths 19 and 21.
A Search for Hadamard Matrices constructed from Williamson Matrices

The material in this chapter has formed the basis of a paper:

Jeffrey Horton, Christos Koukouvinos and Jennifer Seberry. A Search for Hadamard Matrices constructed from Williamson Matrices. (submitted)

8.1 Introduction

An Hadamard matrix $H$ of order $n$ has elements $±1$ and satisfies $HH^T = nI_n$. These matrices are used extensively in coding and communications [see Seberry and Yamada [152]]. The order of an Hadamard matrix is $n \equiv 0 \mod 4$. The first unsolved case is order 428. That is, neither the existence nor non-existence of Hadamard matrices for this order has so far been established.

We use Williamson’s construction as the basis of an algorithm to construct a distributed computer search for new Hadamard matrices. The theory of Williamson’s construction is briefly described in Section 8.2. Before presenting a description of the search algorithm, Section 8.3 discusses two equivalent representations of Hadamard matrices derived from Williamson’s construction; the results of our search is presented using both representations. The implementation of the search algorithm is presented in Section 8.4, some remarks on the time required to perform various searches may be found in Section 8.5, and the results of the search are described in Section 8.6.

A printed version of the source code for the search program is not included: at slightly fewer than 15,000 lines of code and comments, it is too large for inclusion in printed form. It is, however, present on an attached disk of supplementary material.
8.2 Hadamard Matrices from Williamson Matrices

Theorem 15 (Williamson [169]) Suppose there exist four \((1,-1)\) matrices \(A, B, C, D\) of order \(n\) which satisfy

\[
XY^T = YX^T, \forall X, Y \in \{A, B, C, D\}
\]

Further, suppose

\[
AA^T + BB^T + CC^T + DD^T = 4nI_n \tag{8.1}
\]

Then

\[
H = \begin{bmatrix}
A & B & C & D \\
-B & A & -D & C \\
-C & D & A & -B \\
-D & -C & B & A
\end{bmatrix}
\tag{8.2}
\]

is an Hadamard matrix of order \(4n\) constructed from a Williamson array.

Let the matrix \(T\) given below be called the shift matrix:

\[
T = \begin{bmatrix}
0 & 1 & 0 & \cdots & 0 \\
0 & 0 & 1 & \cdots & 0 \\
& & & \cdots & \cdots \\
0 & 0 & 0 & \cdots & 1 \\
1 & 0 & 0 & \cdots & 0
\end{bmatrix} \tag{8.3}
\]

and note

\[
T^n = I, (T^i)^T = T^{n-i} \tag{8.4}
\]

If \(n\) is odd, \(T\) is the matrix representation of the \(n\)th root of unity \(\omega, \omega^n = 1\).

Let

\[
\begin{cases}
A = \sum_{i=0}^{n-1} a_i T^i, & a_i = \pm 1, a_{n-i} = a_i \\
B = \sum_{i=0}^{n-1} b_i T^i, & b_i = \pm 1, b_{n-i} = b_i \\
C = \sum_{i=0}^{n-1} c_i T^i, & c_i = \pm 1, c_{n-i} = c_i \\
D = \sum_{i=0}^{n-1} d_i T^i, & d_i = \pm 1, d_{n-i} = d_i
\end{cases} \tag{8.5}
\]

Then matrices \(A, B, C, D\) may be represented as polynomials. The requirement that \(x_{n-i} = x_i, x \in \{a, b, c, d\}\) forces the matrices \(A, B, C, D\) to be symmetric.
8.2. Hadamard Matrices from Williamson Matrices

Since $A, B, C, D$ are symmetric, (8.1) becomes:

$$A^2 + B^2 + C^2 + D^2 = 4nI_n$$

and the relation $XY^T = YX^T$ becomes $XY = YX$ which is true for polynomials.

**Definition 11** Williamson matrices are $(1, -1)$ symmetric circulant matrices. As a consequence of being symmetric and circulant they commute in pairs.

We use the following theorem of Williamson’s as the motivator for our search algorithm:

**Theorem 16 (Williamson [169])** If there exist solutions to the equations

$$\mu_i = 1 + 2 \sum_{j=1}^{s} t_{ij}(\omega^j + \omega^{n-j}), i = 1, 2, 3, 4$$

(8.6)

where $s = \frac{1}{2}(n - 1), \omega$ is a $n$th root of unity, exactly one of $t_{1j}, t_{2j}, t_{3j}, t_{4j}$ is nonzero and equals $\pm 1$ for each $1 \leq j \leq s$, and

$$\mu_1^2 + \mu_2^2 + \mu_3^2 + \mu_4^2 = 4n$$

then there exist solutions to the equations:

$$\begin{cases}
A = \sum_{i=0}^{n-1} a_i T^i, & a_0 = 1, a_i = a_{n-i} = \pm 1 \\
B = \sum_{i=0}^{n-1} b_i T^i, & b_0 = 1, b_i = b_{n-i} = \pm 1 \\
C = \sum_{i=0}^{n-1} c_i T^i, & c_0 = 1, c_i = c_{n-i} = \pm 1 \\
D = \sum_{i=0}^{n-1} d_i T^i, & d_0 = 1, d_i = d_{n-i} = \pm 1
\end{cases}$$

(8.7)

That is, there exists an Hadamard matrix of order $4n$.

In matrix form, $\omega^j + \omega^{n-j}$ is represented as $T^j + T^{n-j}$. Since these are symmetric, we write

$$\omega_j = \omega^j + \omega^{n-j}$$

We assume

$$\begin{cases}
2A = -\mu_1 + \mu_2 + \mu_3 + \mu_4 = 2\left(1 + \sum_{j=1}^{s} (-t_{1j} + t_{2j} + t_{3j} + t_{4j})(\omega^j + \omega^{n-j})\right) \\
2B = \mu_1 - \mu_2 + \mu_3 + \mu_4 = 2\left(1 + \sum_{j=1}^{s} (t_{1j} - t_{2j} + t_{3j} + t_{4j})(\omega^j + \omega^{n-j})\right) \\
2C = \mu_1 + \mu_2 - \mu_3 + \mu_4 = 2\left(1 + \sum_{j=1}^{s} (t_{1j} + t_{2j} - t_{3j} + t_{4j})(\omega^j + \omega^{n-j})\right) \\
2D = \mu_1 + \mu_2 + \mu_3 - \mu_4 = 2\left(1 + \sum_{j=1}^{s} (t_{1j} + t_{2j} + t_{3j} - t_{4j})(\omega^j + \omega^{n-j})\right)
\end{cases}$$

(8.8)
If \( t_{ij} \neq 0 \) then the coefficient of \( \omega^j + \omega^{n-j} \) in \( x_i \) is different from the coefficient of \( \omega^j + \omega^{n-j} \) in the other equations. Thus with (8.6),

\[
\begin{align*}
    a_j &= -t_{1j} + t_{2j} + t_{3j} + t_{4j} \\
    b_j &= t_{1j} - t_{2j} + t_{3j} + t_{4j} \\
    c_j &= t_{1j} + t_{2j} - t_{3j} + t_{4j} \\
    d_j &= t_{1j} + t_{2j} + t_{3j} - t_{4j}
\end{align*}
\]

From here we should assume that \( n \) is odd. Let

\[ A = P_1 - N_1 \]  

(8.9)

where \( P_1 \) is the sum of positive terms in \( A \) and \( N_1 \) is the sum of the negative terms in \( A \). So

\[
P_1 = \sum_j a_j T^j, a_j = 1; \quad -N_1 = \sum_j a_j T^j, a_j = -1;
\]

(8.10)

In the same way we write

\[
B = P_2 - N_2; \quad C = P_3 - N_3; \quad D = P_4 - N_4
\]

(8.11)

Since \( a_0 = 1 \), and \( a_{n-i} = a_i, i = 1, \ldots, n-1 \), the positive terms except for \( a_0 \) occur in pairs, hence \( p_1 \), the number of terms in \( P_1 \), is an odd number. Similarly, \( p_2, p_3, p_4 \) are odd numbers.

Let us write

\[ J = I + T + T^2 + \cdots + T^{n-1} \]

(8.12)

Then

\[ P_i + N_i = J, \quad i = 1, 2, 3, 4 \]

(8.13)

Using (8.9) and (8.11) and substituting \( N_i = J - P_i \) in (8.1) we have

\[ (2P_1 - J)^2 + (2P_2 - J)^2 + (2P_3 - J)^2 + (2P_4 - J)^2 = 4n \]

(8.14)

Since \( T^j J = J \) for all \( j \), we have \( P_i J = p_i J \) and \( J^2 = nJ \). So (8.14) takes the form

\[ 4(P_1^2 + P_2^2 + P_3^2 + P_4^2) - 4(p_1 + p_2 + p_3 + p_4) J + 4n J = 4nI_n \]

(8.15)
8.2. Hadamard Matrices from Williamson Matrices

Divided by 4 becomes

\[ P_1^2 + P_2^2 + P_3^2 + P_4^2 = (p_1 + p_2 + p_3 + p_4 - n)J + nI_n \]  \hspace{1cm} (8.16)

Since each \( p_i \) is odd and \( n \) is odd, \( p_1 + p_2 + p_3 + p_4 - n \) on the right of (8.16) must be an odd number. In \( P_i^2 = (\Sigma T^k)^2 \), \( k \) in a subset of \( 0, 1, \cdots, n - 1 \). We have \( P_i^2 \equiv \Sigma T^{2k} \pmod 2 \). Hence, every \( T^k = (T^k)^2 \), to appear with an odd coefficient on the right of (8.16), must occur in exactly one or three of \( P_1, P_2, P_3, P_4 \).

**Remark 2** The solutions for (8.6) are independent of the particular root \( \omega \), so if \( n \) as defined by (8.1) is prime, we can choose \( \omega \) so that the first \( \mu \) having any \( \omega_j \) assigned has \( \omega_1 \). Since the equations are true for all roots of unity \( \omega \), they are also true for \( \omega = 1 \).

**Theorem 17 (Williamson [169])** Let \( n \) be odd, and matrices \( A, B, C, D \) satisfy (8.1) and (8.5), suppose \( a_0 = b_0 = c_0 = d_0 \), then exactly three of \( a_j, b_j, c_j, d_j, 1 \leq j \leq n - 1 \), have the same sign.

**Example 1** Let \( n = 9; 4 \times 9 = 36 = 1^2 + 1^2 + 3^2 + 5^2 \). So

\[
\begin{align*}
\mu_1 &= 1 \\
\mu_2 &= 1 \\
\mu_3 &= 1 - 2(\omega^2 + \omega^7) \\
&= 1 - 2\omega_2 \\
\mu_4 &= 1 + 2(\omega + \omega^8) + 2(\omega^3 + \omega^6) - 2(\omega^4 + \omega^5) \\
&= 1 + 2\omega_1 + 2\omega_3 - 2\omega_4
\end{align*}
\]

Now the complex conjugate of \( \mu_i \) is \( \bar{\mu}_i = \mu_i \) and

\[
\begin{align*}
\mu_1^2 + \mu_2^2 + \mu_3^2 + \mu_4^2 &= 1^2 + 1^2 + (1 - 2\omega^2 - 2\omega^7)^2 + (1 + 2\omega + 2\omega^8 + 2\omega^3 + 2\omega^6 - 2\omega^4 - 2\omega^5)^2 \\
&= 1^2 + 1^2 + (9 + 4\omega^4 + 4\omega^5 - 4\omega^2 - 4\omega^7) + (25 + 4\omega^2 + 4\omega^7 - 4\omega^4 - 4\omega^5) \\
&= 1^2 + 1^2 + 3^2 + 5^2
\end{align*}
\]

If \( \omega = 1 \)

\[ \mu_1 = \mu_2 = 1; \; \mu_3 = -3; \; \mu_4 = 5 \]
We form $x_i$ of (8.8)

\begin{align*}
2A &= 2(1 + \omega_1 - \omega_2 + \omega_3 - \omega_4) \\
2B &= 2(1 + \omega_1 - \omega_2 + \omega_3 - \omega_4) \\
2C &= 2(1 + \omega_1 + \omega_2 + \omega_3 - \omega_4) \\
2D &= 2(1 - \omega_1 - \omega_2 - \omega_3 + \omega_4)
\end{align*}

Then writing $S_i = T^i + T^{n-i}$

\begin{align*}
A &= I + S_1 - S_2 + S_3 - S_4 \\
B &= I + S_1 - S_2 + S_3 - S_4 \\
C &= I + S_1 + S_2 + S_3 - S_4 \\
D &= I - S_1 - S_2 - S_3 + S_4
\end{align*}

and using them as (8.2) gives Hadamard matrix of order 36.

**Theorem 18 (Whiteman [172])** Let $q$ be a prime power $\equiv 1 (mod \ 4)$ and put $n = (q + 1)/2$. Let $\gamma$ be a primitive root of $GF(q^2)$. Put $\gamma = ax + b$, where $a, b \in GF(q)$ and define

\begin{align*}
ak &= \chi(a), b_k = \chi(b) \quad (8.17)
\end{align*}

Then the sums

\begin{align*}
f(\xi) &= \sum_{i=0}^{n-1} a_4 i \xi^i, \quad g(\xi) = \sum_{i=0}^{n-1} b_4 i \xi^i \quad (8.18)
\end{align*}

satisfy the identity

\begin{align*}
f^2(\xi) + g^2(\xi) &= q \quad (8.19)
\end{align*}

for each nth root of unity $\xi$ including $\xi = 1$.

Note that when $\xi = 1$ the identity (8.19) reduces to the classical result that every prime $\equiv 1 (mod \ 4)$ is representable as the sum of two squares of integers.

**Corollary 1** Let $q = 2n - 1$ be a prime power $\equiv 1 (mod \ 4)$. Put

\begin{align*}
\psi_1(\xi) &= 1 + f(\xi), \quad \psi_2(\xi) = 1 - f(\xi), \quad \psi_3(\xi) = \psi_4(\xi) = g(\xi)
\end{align*}

where $f(\xi)$ and $g(\xi)$ are the polynomials defined by (8.18). Then the identity

\begin{align*}
\psi_1^2(\xi) + \psi_2^2(\xi) + \psi_3^2(\xi) + \psi_4^2(\xi) &= 4n
\end{align*}

is satisfied for each nth root of unity $\xi$ including $\xi = 1$. 
Theorem 19 (Turyn’s theorem [171], proof by Whiteman [172]) Let \( q \) be a prime power, \( q = 2n - 1 \equiv 1 \text{ mod } 4 \). Then there exists a Williamson matrix of order \( 4n \) in which \( A \) and \( B \) agree only on the main diagonal and moreover, \( C = D \).

We have \( a_0 = 1, b_0 = 1 \). The successive elements in the first row of \( A \) are \( 1, a_4, a_8, \ldots, a_{4(n-1)} \). The successive elements in the first row of \( B \) are \( 1, -a_4, -a_8, \ldots, -a_{4(n-1)} \). The successive elements in the first rows of \( C \) and \( D \) are \( 1, b_4, b_8, \ldots, b_{4(n-1)} \). The matrices \( A, B, C, D \) are circulants.

8.3 The representation of Hadamard Matrices

The relationship between two current methods for classifying Williamson matrices, the Williamson decomposition of \( 4n \) into four squares, \( s_1^2 + s_2^2 + s_3^2 + s_4^2 = 4n \), and the row sums \( m_1, m_2, m_3, m_4 \) of the four Williamson matrices \( A, B, C, D \), is now discussed.

Lemma 6 Let the Williamson decomposition into four squares be \( s_1^2 + s_2^2 + s_3^2 + s_4^2 = 4n \). Further, let the row sums of the four Williamson matrices \( A, B, C, D \) be \( m_1, m_2, m_3 \) and \( m_4 \). Let

\[
M = \frac{1}{2} \begin{bmatrix}
-1 & 1 & 1 & 1 \\
1 & -1 & 1 & 1 \\
1 & 1 & -1 & 1 \\
1 & 1 & 1 & -1
\end{bmatrix}, \quad \xi = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{bmatrix}, \quad m = \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \end{bmatrix}
\]

Then

\[
s_1^2 + s_2^2 + s_3^2 + s_4^2 = 4n \iff m_1^2 + m_2^2 + m_3^2 + m_4^2 = 4n
\]

and

\[
M\xi = w \iff Mm = \xi
\]

Proof. From (8.6) we have, using the root \( \omega = 1 \), a decomposition with

\[
s_i = \mu_i = 1 + 4 \sum_{j=1}^{s} t_{ij}, \quad i = 1, 2, 3, 4.
\]

By Williamson’s assumption condition,

\[
s_1^2 + s_2^2 + s_3^2 + s_4^2 = 4n.
\]
8.4. Search Method

On the other hand,

\[ m_1 = \sum_{j=1}^{n} a_j \]

\[ = 1 - 2 \sum_{j=1}^{n-1} t_{1j} + 2 \sum_{j=1}^{n-1} t_{2j} + 2 \sum_{j=1}^{n-1} t_{3j} + 2 \sum_{j=1}^{n-1} t_{4j} \]

\[ = 1 - \frac{1}{2} (s_1 - 1) + \frac{1}{2} (s_2 - 1) + \frac{1}{2} (s_3 - 1) + \frac{1}{2} (s_4 - 1) \]

\[ = \frac{1}{2} (-s_1 + s_2 + s_3 + s_4) \]

Similarly,

\[ m_2 = \frac{1}{2} (s_1 - s_2 + s_3 + s_4) \]
\[ m_3 = \frac{1}{2} (s_1 + s_2 - s_3 + s_4) \]
\[ m_4 = \frac{1}{2} (s_1 + s_2 + s_3 - s_4) \]

and \( M_5 = m \). Inverting we have, as \( M^{-1} = M \), \( Mm = s \). It is easy to check that

\[ m_1^2 + m_2^2 + m_3^2 + m_4^2 = s_1^2 + s_2^2 + s_3^2 + s_4^2 = 4n. \]

8.4 Search Method

8.4.1 Introduction

The basic search method is to examine all possible combinations of \( \omega_j, 1 \leq j \leq \frac{1}{2} (n - 1) \) for each \( \mu_i, i = 1, 2, 3, 4 \), testing each set of \( \mu \) so generated to see if it satisfies Williamson's condition and can be used to form an Hadamard matrix of order \( 4n \). This search method is documented in more detail in the following sections.

First experiences with a search program based on this technique indicated that the size of the search space rapidly grew so large that it was beyond the capabilities of any single available computer to perform a search for Hadamard matrices of order 148 within a reasonable time, which was defined as weeks or months rather than years; this was not unexpected. The most powerful computer available at the time for performing the search was a Silicon Graphics machine with 4 180MHz R10000 microprocessors.
A subsequent version of the program used a more distributed approach — a client program running on a remote computer requests a server program to supply subproblems for processing by the client. Any solutions found by the client are reported to the server. Very little work is done by the server; most work performed will be shared amongst the clients.

This has the advantage that if the clients can work largely independently and do not require to communicate often or extensively with the server to perform work, then large amounts of work can be accomplished in a fraction of the time required for the same amount of work to be performed by a single computer. A doubling of the number of clients results in halving the time taken to perform a search, for example, if the problem being considered is suitable.

There are a number of software tools and packages available to make the construction of systems to perform a distributed computation more straightforward. These include **PVM** or “Parallel Virtual Machine” [177], and **MPI** or “Message Passing Interface” [178]. Both PVM and MPI have been ported to a variety of different computer environments; PVM implementations are interoperable, which is not necessarily the case for different implementations of MPI at the present time. PVM and MPI both provide facilities allowing the exchange of messages between tasks participating in a particular computation.

However, it was decided not to use one of these packages, but instead to use lower-level networking APIs directly to perform any required communications, for several reasons:

1. Many of the computers potentially available locally to perform the computation were Macintosh computers or computers running some form of UNIX. We are not aware of any publicly available package like PVM or MPI that was available for both UNIX-like systems and the Macintosh and able to interoperate between different implementations\(^1\). It was not particularly difficult, however, to produce a client program which could be compiled and run on a wide variety of UNIX-like systems and which used sockets to perform communication.

\(^1\)Project Appleseed [179] has been working with a partial implementation of MPI running on a cluster of Macintosh computers. However, their implementation currently uses the AppleTalk networking protocol and so would be unlikely to interoperate easily, if at all, with any UNIX-based implementation of MPI.
Search Method

via TCP/IP [180], and another which operated on the Macintosh using Open Transport to implement TCP/IP communications [181]. Differences in byte ordering between computer architectures is another problem which can cause a difficulty for client-server systems, and which is typically handled by a package such as PVM. In this case all data was converted to and from a simple platform-independent form for exchange between machines.

2. It was desired to produce a system which could potentially involve computers not located on the local area network, and there was some concern that available systems such as PVM and MPI might not be able to incorporate these remote computers reliably and robustly, if at all.

3. Computers located behind a firewall may have difficulty in participating in a distributed search such as this if the firewall has been configured so that only connections from or to particular ports are permitted. For example, a firewall might be configured so that outgoing connections are permitted only via port 80, which is the network port on which WWW servers usually listen for connections from WWW browsers. Communication between WWW servers and browsers takes place using a protocol known as Hypertext Transfer Protocol, or HTTP. Using a low-level networking API for communication between a client and server performing a distributed computation enables such environmental restrictions to be overcome — for a suitably simple protocol, all transactions between client and server can take place encapsulated as HTTP. It was unnecessary to implement such sophisticated features to successfully perform the searches described here, however.

4. It was not desirable that participation in the search require any software to be installed in addition to the client program for performing the search. The need for a remote site to install extra software in addition to the search client itself may affect the willingness of that site to participate in the search.

Searches for Hadamard matrices of all orders up to and including order 148 have been performed using Williamson’s method implemented by a client/server system. Towards the end of an initial search of order 148, 37 computers were involved, 20
270MHz Ultra 5 computers from Sun Microsystems, and 17 333MHz iMacs from Apple Computer. No computers not available on the local area network were employed in the initial search. However, a subsequent search performed to verify results utilised 35 350MHz Pentium-II computers at the University of Newcastle in addition to 30 local Ultra 5 computers.

The details of the implementation of Williamson’s method within the framework of a client/server system are discussed in the following sections.

8.4.2 Decompose $4n$ into sum-of-squares representation

The first step in performing a search is to decompose $4n$ into all possible sums-of-squares representations. Observing the form of (8.6), we see that when $\omega = 1$ each $\mu_i$ satisfies:

$$|\mu_i| \equiv 1 \mod 4, \mu_i > 0; \text{ or}$$
$$|\mu_i| \equiv 3 \mod 4, \mu_i < 0. \tag{8.20}$$

For example, the possible decompositions for 148 are:

- $1, 1, 5, 11$
- $1, 7, 7, 7$
- $3, 3, 3, 11$
- $3, 3, 7, 9$
- $5, 5, 7, 7$

In the sections to follow, we write $\omega_{\text{sub}}$ to indicate some $\omega_k = \omega^k + \omega^{n-k}$ for $1 \leq k \leq \frac{1}{2}(n - 1)$ when it is necessary to distinguish from an $n$th root of unity, $\omega$.

8.4.3 Decide on the number of $\omega_{\text{sub}}$ assigned to each $\mu$

The next step is to assign a number of $\omega_{\text{sub}}$ to each $\mu$. Using (8.20), we see that if $|\mu_i| \equiv 1 \mod 4$, then of the $\omega_{\text{sub}}$ contributing to $\mu_i$, the number being added to $\mu_i$ will always be $\frac{|\mu_i| - 1}{4}$ greater than the number of $\omega_{\text{sub}}$ that are subtracted. A similar condition can be derived for $|\mu_i| \equiv 3 \mod 4$. These $\omega_{\text{sub}}$ are termed “fixed”; others are “floating” and always occur in pairs, one added and the other subtracted. These conditions are enforced to help limit the size of the space to be searched.
All possible permutations of the number of floating $\omega_{\text{sub}}$ are assigned to each $\mu$ over the course of the search of a particular sum-of-squares representation, subject to certain restrictions that are useful for reducing the size of the space to be searched:

1. The number of $\omega_{\text{sub}}$ assigned to $\mu_i$ must be greater than or equal to the number of $\omega_{\text{sub}}$ assigned to $\mu_j$ where $j < i$ and $\mu_i$ and $\mu_j$ correspond to the same value in the sum-of-squares decomposition. We may apply this condition because for the purposes of testing the set of $\mu$ to see if Williamson's condition is satisfied, $\mu_i$ and $\mu_j$ are interchangeable, and it is desirable to perform the test only once rather than twice. This may be extended further if more than two $\mu$ have the same value in the sum-of-squares decomposition.

2. If $n$ is prime, then we may always place $\omega_1$ in the first $\mu$ to which any $\omega_{\text{sub}}$ are assigned. This corresponds to solving the set of $\mu$ for some $n$th root of unity, $\omega^j$, such that $\omega_1$ is present in the first $\mu$ to which any $\omega_{\text{sub}}$ are assigned. Furthermore, if there are $\omega_{\text{sub}}$ both added and subtracted from this $\mu$, we may either subtract or add $\omega_1$; we do not need to check both. If this condition is in force, then condition 1 is not applied in the case of the $\mu$ to which $\omega_1$ is assigned, but remains applicable for other $\mu$ corresponding to the same value from the sum-of-squares decomposition. Enforcing this condition can greatly reduce the size of the space to be searched: for example, applying this condition for searching for Hadamard matrices of size 148 reduces the size of the space to be searched to 37% of its size were this condition not to be enforced (reducing from about 32,387,862,644,280 to 12,062,406,963,464)\(^2\).

For each permutation of floating $\omega_{\text{sub}}$ that is generated, we must assign specific identities to each $\omega_{\text{sub}}$ and evaluate Williamson's condition.

### 8.4.4 Assign specific identities to each $\omega_{\text{sub}}$

We must now assign specific identities to each $\omega_{\text{sub}}$ so that Williamson's condition may be tested.

\(^2\)We would have achieved an even greater reduction in the size of the search space had we not been checking for solutions by both adding and subtracting $\omega_1$ where this option was available. In this case, the size of space to be searched is less than half of the above figure.
8.4. Search Method

Let the number of $\omega_{\text{sub}}$ added to $\mu_i$ be represented by $c_{2i-1}$ and the number of $\omega_{\text{sub}}$ subtracted from $\mu_i$ by $c_{2i}$. $S_{2i-1}$ is the set of $\omega_{\text{sub}}$ added to $\mu_i$ and $S_{2i}$ is the set of $\omega_{\text{sub}}$ subtracted from $\mu_i$. That is, there are eight sets $S$, two for each $\mu$. Some of these sets $S$ may be empty.

$$\mu_i = 1 + 2 \sum_{\forall j \in S_{2i-1}} \omega_j - 2 \sum_{\forall j \in S_{2i}} \omega_j$$

Dividing $\omega_{\text{sub}}$ into two groups, one added to a $\mu$ and the other subtracted, helps to simplify the procedure for iterating over all possible combinations of $\omega_{\text{sub}}$.

The sets $S_i$ are formed by choosing $c_i$ elements from the set of $\omega_{\text{sub}}$ not already allocated to an $S_j$, $j < i$. Recalling that $s = \frac{1}{2}(n-1)$, $S_{T,0}$ is defined as:

$$S_{T,0} = \{\omega_1, \omega_2, \omega_3, \ldots, \omega_s\}.$$ 

$S_{T,i}$ is defined as:

$$S_{T,i} = S_{T,i-1} - S_{i-1}, i = 1, \ldots, 8. \quad (8.21)$$

For convenience, we say that:

$$S_0 = \emptyset$$

Williamson's condition may be tested once $S_1, \ldots, S_8$ have been generated. All possible combinations of $c_i$ elements from $S_{T,i}$ are examined; once the combinations are exhausted, the next combination for $S_{i-1}$ is generated. The process is illustrated by the small segment of pseudocode shown in Figure 8.1.

So it should be easy to see that the number of tests of Williamson's condition for a particular set of $c_1, \ldots, c_8$ can be calculated as follows:

$$\text{Evaluations} = \prod_{i=1}^{8} \binom{|S_{T,i}|}{c_i} \quad (8.22)$$

Usually, however, the total number of evaluations performed will be less than this, for two reasons:

1. If condition 2 from Section 8.4.3 is applied, we choose one fewer $\omega_{\text{sub}}$ for the set $S$ in which $\omega_1$ is to appear.

2. If $\mu_i$ and $\mu_j$, $i < j$ correspond to the same value in the sum-of-squares decomposition of $4n$ and have the same number of $\omega_{\text{sub}}$ assigned, then we may require
\[ j := 1; \]
\[
\text{do} \\
\text{for } k \text{ from } j \text{ to } 8 \\
\text{popu}
\]
\[ \text{late } S_{T,k} \text{ from } S_{T,k-1} \text{ and } S_{k-1} \text{ using (8.21);} \]
\[ \text{generate combination } S_k \text{ by choosing } c_k \text{ elements from } S_{T,k}; \]
\[ \text{Test Williamson Condition using } S_1, \ldots, S_8 \text{ to generate } \mu_1, \ldots, \mu_4; \]
\[ j := 8; \]
\[ g := \text{false}; \]
\[
\text{while } ((j > 0) \text{ and } (g = \text{false})) \\
\text{generate new combination } S_j \text{ using } c_j \text{ elements from } S_{T,j} \\
\text{if successful} \\
\quad g := \text{true}; \\
\quad j := j + 1; \\
\text{else} \\
\quad j := j - 1; \\
\text{while } (j > 0); \]

Figure 8.1: Segment of pseudocode illustrating generation of combinations for testing Williamson's condition.

that if \( \omega_x \) is the \( \omega_{\text{sub}} \) of smallest subscript assigned to \( \mu_i \) and \( \omega_y \) has the smallest subscript assigned to \( \mu_j \), that \( x < y \). Otherwise, work will be repeated when \( \mu_i \) replicates a sequence that had previously occurred in \( \mu_j \). Enforcing this condition ensures that no repetition takes place and reduces the size of the search space slightly. The reduction is considerable but unfortunately not as substantial as that for applying condition 2 from Section 8.4.3.

### 8.4.5 Dividing up the work for distribution

The easiest way in which to divide up the work for distribution by the server to the clients would be for the clients to perform all the work described in Section 8.4.4, with the server being responsible for deciding only the number of \( \omega_{\text{sub}} \) being assigned to each \( \mu \) for a particular sum-of-squares decomposition, as described in Sections 8.4.2 and 8.4.3.

However, this not only burdens the clients with far too much work but also limits the number of clients which may be involved in the search. It is best to assign the
clients an amount of work that would take a small fraction of the total time required to perform the whole search, but not so small a fraction that network communication overheads become excessive. That is, it is desirable that the clients seldom require to communicate with the server. A few hours' work is usually a good target to aim for.

Catering for the varying abilities of computers is also useful. For example, if one computer is very much older and slower than some others participating in the search, then it may be useful to assign it a smaller amount of work than that assigned to faster computers so that the slower machines may still make a contribution to the search. However, such refinements, while desirable, were not implemented for the performance of this search.

The obvious manner in which to reduce the amount of work performed by the clients to a reasonable level was to make the server perform part of the work described in Section 8.4.4. The server performs no evaluations itself, but would choose sets $S_1, \ldots, S_i$, for some $i < 8$. The client would evaluate all the possibilities for the choice of the remaining sets $S_{i+1}, \ldots, S_8$.

Unfortunately, this does not yield subproblems with an even division of work. Some subproblems will require clients to perform many more evaluations than others. The server decides what value $i$ should take by estimating the amount of work involved in a subproblem using a modification of (8.22):

$$\text{Evaluations performed by client} = \prod_{j=i+1}^{8} \left( \frac{|S_{T,j}|}{c_j} \right). \quad (8.23)$$

Two constants $S_{\text{min}}$ and $S_{\text{max}}$ must be specified to the server: a subproblem is of acceptable size if its size lies between the two limits.

This can result in subproblems which are very small if $S_{\text{max}}$ is not set sufficiently high. Many small subproblems also mean that the server is performing very much more work in terms of allocating and transmitting a work unit than is desirable. It can be difficult to determine that $S_{\text{max}}$ is not sufficiently high when starting the search. There are a number of possible solutions to the problem; fortunately none of these were required for the searches described in this paper:

1. The server can allocate multiple small subproblems to a client looking for work;

2. Instead of just one server, a master server and a number of sub-servers or proxies could be designed. Clients could obtain work directly from the master server.
or from a proxy; similarly, results could be reported to the master or a proxy. Proxies would synchronize with the master server on a daily or weekly basis, depending on the size of the problem;

3. Implement a different scheme of partitioning the work that gives more control over the size of the subproblems. One way of changing the partitioning of the work would be to make the server responsible for choosing say half of the $\omega_{\text{sub}}$ for one of the sets $S$, and the client responsible for choosing the other half and performing all necessary evaluations. In addition to being rather more complicated, this scheme as well would have had some problems regarding the even division of work, perhaps not to the same extent as the scheme we employed, however.

The server must also keep a record of each subproblem it allocates, so that should any client fail to complete a subproblem, perhaps because the client software was deleted from the computer on which it was running along with any records of subproblems allocated to that client, it will still be possible to complete the computation. The implementation described here assigns old, as yet unsolved subproblems in response to client requests for work once there were no new subproblems to be assigned. This ensures that all the work required to solve the complete problem is actually performed.

### 8.4.6 An update on methods of dividing the work for distribution

When searches were performed, it was noticed that there were a large number of subproblems that involved individually very little work. These could be completed very quickly, and resulted in a large number of reports of completed problems and requests for new problems being handled by the server over a short space of time.

Several modifications were made to the server and client to change this behaviour, with the aim of being able to support a much larger number of client machines whose contacts with the server are much better distributed over time.

These modifications are as follows:

- To even out the amount of work assigned to each client, the server assigns multiple subproblems to a client so that each client has a certain minimum amount of work to perform.
The server was initially a multithreaded design. Most of the threads handle communications with the clients. Network communication, especially to remote sites, can be quite slow, so multiple threads enable more than one client to be handled simultaneously if required. We considered a multithreaded design to be superior to a multiprocess design in the case of the server, because there is a considerable amount of shared state between the threads. It would be more difficult to share this state between multiple processes. An additional thread was added that maintained a queue of pre-calculated problems ready for quick assignment to clients. This same thread also maintained a queue of completed problems waiting to be written to the problem store. This improves the ability of the server to handle bursts of requests from clients with a minimal amount of computation, synchronization or disk access. This is a useful feature if many clients are to participate in the search.

### 8.4.7 A note on process control

Process control on remote machines is a challenging problem. Processes must be initially started, re-started if stopped, and on occasion it is useful to check that processes are still running or send a process a signal to trigger some behaviour.

Control of processes running on machines running some variant of UNIX was initially handled using very simple bash shell scripts using `rsh` for communication with remote machines. These simple scripts were later replaced with more sophisticated, flexible scripts written using the *Tool Command Language* [182, 183], Tcl, and also using `rsh` for communication with remote machines. This represented a considerable improvement over the earlier simple shell scripts. A script has also been prepared to automate interaction with the `telnet` command. This script can be used to access multiple remote machines quickly and easily in the absence of `rsh`. The script was written in *Expect* [184], a Tcl extension.

Tcl also includes simple facilities for working with TCP/IP sockets, which was of use in stress-testing the server.

It is unfortunate that process control for Macintosh computers is practically non-existent. The simplest and most effective method was to check each machine manually. However, we believe better and more effective methods may exist. The small number
of Macintoshes available and their relatively poor performance compared to computers running UNIX discouraged excessive experimentation.

8.5 Time required for the performance of the searches

Searching for Hadamard matrices using Williamson’s method as previously described for matrices of order 100 is very quick — a complete search can be performed by a single 270MHz Sun Microsystems UltraS in less than 15 minutes; the exact time taken will depend on the options employed to limit the size of the search space.

Searches for Hadamard matrices of orders higher than 100 utilised multiple machines. This makes more than a rough estimate of the amount of time required to perform the search impossible: over the course of a long-running search, computers will crash or be rebooted, or perhaps might not have facilities for reporting the amount of CPU time consumed by a process.\(^3\)

Performing a complete search for matrices of order 132 required between two and three days’ running time on 20 UltraS computers. Performing the search for matrices of order 148 required over a month utilising initially only 20 UltraS computers; 17 Macintosh iMac computers became available towards the end of the search that enabled it to be completed in slightly over a month. A total of 10,698,065,244,724 possible sequences were tested with Williamson’s condition. The search was subsequently repeated, requiring slightly over two weeks’ time on 20 UltraS computers and 17 Macintosh iMac computers, and testing 12,062,406,963,464 possibilities.\(^4\) A more recent repetition of the search used 30 UltraS computers and 35 350MHz Pentium-II computers, checking 5,772,437,757,944 possibilities in less than a week.\(^5\)

The performance of the search code was rather poor on the Macintoshes involved in the search. Given the specifications of the machines, it was expected that the code would perform substantially better than it did in practice, when compared with the performance of the UltraS machines. However, the performance of the search code

---

\(^3\)Macintoshes running the Macintosh operating system, rather than some form of UNIX or Linux, do not have the capability to report the amount of CPU time consumed by a process.

\(^4\)This differs from the previous count because the second search was not excluding duplicate sequences as per Condition 2 of Section 8.4.4.

\(^5\)This count differs from previous searches as a result of our implementation of our earlier remarks regarding choosing \(\omega\).
when run on a Macintosh running Linux was substantially better than when run on the same computer using the standard Macintosh operating system.

It is estimated that performing a complete search using the techniques described here for matrices of order 164 would require the participation of about 300 computers for between one and two months. This is not viewed as being infeasible, now that modifications have been made to the search program to better cope with the many small subproblems that were generated during the performance of the search for matrices of order 148. Modifications made to date are described in Section 8.4.6.

8.6 Search Results

Unfortunately, no new matrices were found as a result of the searches run so far. However, we are able to provide independent verification of results from previous searches. This is considered of utility since some previous searches, such as that conducted by Sawade [163], for example, failed to reveal all solutions that are now known for the order searched, in that case, order 100. In particular, we provide verification of results reported by Dokovic [168, 164] for orders 100, 140 and 148. Results for order 100 are also verified by Christos Koukouvinos.

Previous searches are summarised in Section 6.2.6. For reference purposes, complete tables of Hadamard matrices derived from Williamson matrices using circulant symmetric \((1,-1)\) matrices in the Williamson array for orders 12 through 148 are presented in Appendix C, using the notation of the Williamson decomposition and the row sums representation. Example 1 illustrates the process of converting from the notation of the Williamson decomposition to the row sums representation. Recall that each row of a solution found in row sums representation represents the first row of one of the circulant symmetric \((1,-1)\) matrices \(A, B, C, D\).
Appendix A

Reproduction of “Computer Viruses: An Introduction”

The material here is a reproduction of “Computer Viruses: An Introduction”, an introductory paper on computer viruses presented at the Australasian Computer Science Conference held at Macquarie University in 1997. This material has been superseded by material in Chapter 2, but is included here for reference.
Abstract

Computer viruses pose a considerable problem for users of personal computers. The recent emergence of macro viruses as a problem of some importance may heighten virus awareness in general. Yet most people have little or no understanding of common anti-virus measures, the varieties of viruses that exist today, and the strategies which they use to accomplish infection and to defeat anti-viruses. It is well-known that the virus problem is most severe for users of IBM PCs and compatibles; however, users of other platforms, such as the Macintosh, should not become complacent — viruses exist for many platforms in varying numbers. The ease with which macro viruses may be written is discussed, and a new virus attack for the Macintosh is presented which closely resembles an attack under DOS for the PC.

Keywords Computer science, computer virus, computer viruses, macro viruses, companion viruses.

1 What is a Computer Virus?

There is some difficulty in producing a definition for the term "computer virus". Dr. Cohen has presented a mathematical definition of a computer virus, which may be roughly expressed as:

A virus is a program that can 'infect' other programs by modifying them to include a possibly evolved version of itself. [4, p. 2]

However, this definition classifies as viruses some things which would not be considered viruses by those working in the anti-virus field. At the same time this definition would not consider as viruses programs that infect another without modifying the target program itself [2], such as companion viruses, discussed in Section 3.

Additionally, the above definition does not convey any need for the "virus" to be able to replicate further once it has infected some other program [2] — and replication is viewed as an important characteristic of a true "computer virus".

A definition which is felt to be more useful when dealing with "real" computer viruses than the above definition or Dr. Cohen's mathematical model is:

We define a computer 'virus' as a self-replicating program that can 'infect' other programs by modifying them or their environment such that a call to an 'infected' program implies a call to a possibly evolved, and in most cases, functionally similar copy of the 'virus'. [19, 2]

The term 'infect' is used with respect to computer viruses in the sense of the definition above throughout the remainder of this document.

It is important to note that a virus is not necessarily malicious — although its presence may have side effects (as a result of the presence of the virus and its activities causing problems with the operating system, user programs and extensions to the standard operating system installed by the user) which are deemed to be undesirable.

Some researchers have been considering the question of viruses that perform useful actions — so-called benevolent viruses. Cohen [4, pp. 15-21] considers briefly the topic of benevolent viruses. He considers as examples:

**Compression viruses** — little-used files are compressed by the virus and uncompressed when required.

**Maintenance viruses** — any virus which would perform maintenance tasks in a computer system, such as updating installed programs.

**Distributed Databases with viruses** — viruses would reproduce on networked computers, performing searches for the virus' originator. Results would be reported back...
by mail, and the virus would clean itself up after a certain time.

The use of viruses in covert distributed data processing (specifically, key cracking on encrypted messages) has been proposed by White [13]. It may be argued, however, that this would not be particularly beneficial to the user whose resources are used by the virus.

Many arguments against the idea of benevolent viruses are presented by Bontchev [2].

2 Worms and Trojan Horses

A worm is an independent program that is able to spread copies of itself or of parts of itself to other computers, commonly across network connections, and these copies are themselves fully functional independent programs, which are capable either of spreading further and/or of communicating with the parent worm (to report back results of some computation, for example).

There is often confusion over the distinction between a worm and a virus. For example, the program that negatively affected the Internet in November 1988 is referred to as a virus ("the Internet Virus") by some [5] and as a worm ("the Internet Worm") by others [11, 12, 10]. Spafford [12] argues that referring to the infection as a "worm" rather than a "virus" is most appropriate.

A notable difference between worm programs (such as the Internet Worm) and viruses is that while a virus may take advantage of network connections to infect other programs (some local area networks are particularly susceptible, as the user is able to interact with programs and data stored on a remote machine as if they were available locally), it is not capable of causing its code to execute on a remote machine. Clearly the well-known worms have been able to cause their programs to be executed on the remote machine which was the worm's target.

The Internet Worm affected Sun 3 and VAX systems running variants of BSD UNIX [11]. Other worms have been created with other networks in mind, such as DECnet [7].

A trojan horse is a program which possesses various intentional undocumented features1 whose effects few users of the software would appreciate were these undocumented features to manifest themselves. Unlike a computer virus, which attaches itself to some other program using any of a number of methods, a trojan horse is a self-contained program. A trojan horse may have functions of use to the user.

Some definitions of "trojan horse" define a computer virus as a replicating trojan horse which inserts a copy of itself into some other program [1].

A trojan horse might install a virus as its intentional undocumented action. Some feel that a program which installs a virus as a result of having been previously infected with the virus is also a trojan horse (having been converted into a trojan horse by the virus) [17]. However, this seems incompatible with the notion of a "trojan horse" being a program which was initially produced with an intentional undocumented feature in place.

3 Varieties of Viruses

There are a number of different ways that viruses use to infect a computer system. The two main types of viruses are:

File Infectors: These are viruses that attach themselves to some form of executable code. There is a variety of ways in which a virus when running might attempt to infect a file. On a DOS-based system, file infectors will commonly attach to .COM or .EXE files, although there are many other kinds of infectable objects.

Boot Sector Infectors: Only discussed in the context of a PC-compatible system. These kinds of viruses infect executable code which is loaded from disk and called when a computer is starting up. There are a number of different pieces of code which may be modified by a virus to infect a system, such as:

- DOS boot sector [floppy disks and hard disks].
- Master Boot Record (MBR) [hard disks only].
- Partition table [hard disks only].

A virus that is capable of spreading by infecting files and by infecting via any code executed at boot time is known as a multipartite virus.

Boot sector viruses are extremely widespread; as a group they are easily the most commonly found variety of virus on PC-compatible systems.

A virus may be direct-action or resident [17]. A direct-action virus is one that when initially executed in the course of normal use of a computer system identifies executable objects for infection and exits once infection has been accomplished. Direct-action viruses may also be referred to as non-resident viruses.

A resident virus is one which installs itself somewhere in memory, and makes arrangements for the virus body in memory to be executed at some future time; the virus may infect files or take other

---

1As opposed to bugs in the program, otherwise known as "unintentional undocumented features".
action (to conceal its presence, for example) at the time it is next executed. For example, some Macintosh viruses if resident in memory will infect an application when that application commences running and performs certain system calls required to initialise the Macintosh Toolbox, which consists of a set of utility functions available to all applications.

Some programs useful to the user are also resident programs — this includes some antivirus programs that monitor computer system operations for actions which may indicate the presence of a virus.

There are some other types of viruses which should be mentioned:

**Macro Viruses**: These are explained in detail in Section 4. The ease with which such a virus may be written is discussed in Section 9.2.

**File System or Cluster Viruses**: Rather than infecting files directly, such a virus modifies directory table information so that the virus is executed first. It would then pass control to the program that the caller really wants so as to avoid rapid detection. "Dir-II" is an example of this variety of virus.[17]

**Kernel Viruses**: These are viruses that target a specific feature of an operating system's kernel (the program(s) that represent the heart of an operating system). [17]

**Companion Viruses**: Companion viruses occur in several varieties [3, 8], the most notable being:

- **Regular Companion**: Creates a file in the same directory as the target of infection but with a filename extension which the operating system chooses to execute before the original file (for example, under DOS a .COM file is executed before a .EXE file with the same name). The attack as described would appear to be highly specific to PCs running DOS.

- **PATH Companion**: Create a file with any executable extension in a directory that is searched for executable files before the directory containing the target of infection (named after the PATH environment variables found in operating systems such as DOS and UNIX).

Obviously, not all of these infection strategies are available on some platforms and operating systems. For example, Macintosh computers do not suffer from companion viruses as described above (although there is a possible way of creating a virus with similar behaviour — see Section 9.1), and also do not appear to be afflicted with viruses of the boot sector variety.

All viruses, with the exception of macro viruses, are platform dependent.

### 4 Macro Viruses

The anti-virus community has been aware for some time now of the potential for virus-writing provided by the scripting (or macro) languages of large software packages such as Microsoft Word and Excel. The idea of macro viruses was first introduced by Highland [6].

However, it is only recently that such viruses have become a problem. These viruses are remarkable for the fact that they infect what are usually thought of as documents. A macro virus may also be platform independent, being capable of spreading on any computer platform supported by the host application. The most well-known and widespread are Microsoft Word viruses written in WordBasic (two examples are Concept and Nuclear. Concept is the most common and widespread of all current macro viruses). WordBasic is a version of the BASIC programming language extended with many commands and functions for use in the word processing environment.

Microsoft Word recognises the existence of two different forms of user file — an ordinary document, and a template. A template is very much like an ordinary document, with the addition that templates may also contain macros written in WordBasic. Ordinary documents may be converted easily into templates, but templates may not be transformed into ordinary documents with the same degree of ease (which makes manual removal of macro viruses rather tedious). Word has a "global template", the so-called "Normal template", which is used by every document created, and is important for the spread of macro viruses — the macros that make up the virus are copied into the Normal template where they are subsequently available to other documents.

There are several ways that a macro virus may be created using WordBasic. Macro viruses in Microsoft Word may exploit the existence of several varieties of specially-named macros within WordBasic [15]:

- The **AutoExec** macro, stored within some global template such as the Normal template, which is executed automatically whenever Word is started.

- "Auto" macros, which run whenever certain user actions take place within Word:
  - **AutoNew** runs when a new document is created.
• Macros named for built-in Word commands (menu options are a popular choice), which are run when the command is selected by the user. An example, commonly used by macro viruses, is FileSaveAs. This macro, if present, would be executed if the user should select the Save As ... option from the File menu. The macro may be said to have redefined the built-in Word command.

A recent Word macro virus, the Outlaw virus, is an example of another way in which macros may be executed. Word permits "shortcut keys" to be assigned (by the user or by a macro) to macros, menu commands and the like. The assigned function is performed when the shortcut key(s) are pressed. The Outlaw virus, instead of using automatic macros or macros named for a built-in Word command, assigns shortcut keys to particular viral macros [20]. In the case of the Outlaw virus, viral macros will be executed whenever the user types a space or presses the 'e' key.

There may be other methods that will assist the execution of viral macros within Word. Word macros may be marked as ExecuteOnly, which means that the macros cannot be easily edited or inspected by a Word user but can only be executed. Some viruses, for example the Nuclear virus, use this method to hinder casual analysis of the viral macros.

A basic Word virus is not difficult to create, requiring just one macro. The macro virus DMV (the so-called "Demonstration Macro Virus", created as a demonstration of macro viruses) features a single macro AutoClose. The Normal template will be infected when a document containing the DMV AutoClose macro is closed. Subsequently, documents are infected as they are closed [15]. Concept, a more complicated macro virus, signals its initial infection of the Normal template, and subsequently will infect any document (to be saved as a Word document or template) with the viral macros when the document is saved using the Save as ... option of the File menu (an example of a virus using a macro named for a Word menu option).

Some macro viruses, such as DMV and Concept do nothing but spread. Some would argue that even this is damaging, in terms of the time required to remove the virus macros from whatever documents have been infected. More damaging actions are certainly possible, however. For example, a virus might delete paragraphs from a document, or rearrange or insert words (the Nuclear virus will append some lines of text to documents when printed at the right time). More sophisticated macro viruses attempt to infect the user's computer with an ordinary variety of virus which infects executable files (the Nuclear virus unsuccessfully attempts to do this), however behaviour such as this is platform dependent.

A virus can also give itself some (limited) protection against being removed by implementing a ToolsMacro macro, which is then executed in place of the corresponding menu option should it be selected by the user. This menu command offers one possible way for macros to be removed from a document, if the environment has not already been infected by a virus. However, if the environment is infected, then the possibility exists that the virus will take some damaging action should this option be selected.

Disabling of the features of Microsoft Word which allow automatic execution of certain macros when documents are opened and closed offers some limited protection, but as macro viruses can be written with macros that mask menu options (macros which cannot be disabled in the same way that automatic macros may) and macros which are executed when certain keys or key combinations are pressed, this is hardly a complete solution. Furthermore, as the execution of "auto" macros is controllable from WordBasic (at least under Word 6), "auto" macros may be re-enabled by viral macros that are executed by some other means.

5 Virus Occurrences

There have been a great many viruses created for many different computer platforms. The PC has by far the largest share of all the viruses in existence (the producers of Dr. Solomon's Anti-Virus Toolkit, a leading anti-virus package, claimed to detect 9417 PC viruses in July 1996 [18]).

Many of these PC viruses are closely related (once a virus becomes available, it is not uncommon to find a number of copycat viruses that differ only slightly from the original appearing, perhaps written by less-skilled virus writers using virus source code; alternatively, the virus author might release a number of viruses with different payloads but sharing common code for infection and anti-anti-virus measures).

Some information has been gathered by Virus Bulletin about PC viruses that have been reported as found over the course of a month for some months. The percentage of the reports made up by the various virus classes for several months of 1996 are shown in Table 1 [21].
### Table 1: % of reports to/collected by Virus Bulletin made up by various classes of virus.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro</td>
<td>21.8</td>
<td>18.4</td>
<td>20.2</td>
<td>34.3</td>
</tr>
<tr>
<td>Boot</td>
<td>62.5</td>
<td>64.5</td>
<td>59.2</td>
<td>53.7</td>
</tr>
<tr>
<td>Multipartite</td>
<td>7.3</td>
<td>10.6</td>
<td>13.9</td>
<td>6.6</td>
</tr>
<tr>
<td>File</td>
<td>8.4</td>
<td>4.8</td>
<td>6.4</td>
<td>4.8</td>
</tr>
<tr>
<td>Other</td>
<td>0.0</td>
<td>0.3</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Unclassified</td>
<td>0.0</td>
<td>1.3</td>
<td>0.4</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Only a very few of the total number of known viruses are responsible for the majority of virus incidents. Some on-access anti-virus products may use this fact to help restrict the number of viruses that a file must be checked for when accessed by a user.

Viruses exist for a number of other personal computer platforms, such as the Amiga and the Macintosh, but the numbers of these viruses are a small fraction of the numbers of viruses available for the PC. The Macintosh, for example, is afflicted with only a few dozen viruses. Viruses written to a file must be checked for when accessed by a user.

There are a variety of strategies which a virus might employ to hinder detection of the virus, and analysis once it has been discovered:

**Stealth:** When initially run, the virus can arrange to intercept disk reads while present in memory. If it detects an attempt to read a section of the disk (such as the boot sector, partition table or master boot record) or a file that has been infected by the virus, it can conceal its presence by returning as a result of the call data with no signs of infection. Some examples of stealth viruses are [16]:

- Members of the “Brain” strain of viruses are stealth floppy boot sector viruses.
- The “512” virus (also known as “Number of the Beast”) is a stealth file infecting virus.

**Polymorphic Viruses:** A polymorphic virus is one that is capable of varying many aspects of its appearance in the hope of avoiding detection. Cohen refers to viruses of this type as “evolutionary” viruses [4, p. 73]. Some of the strategies which might be employed by a polymorphic virus are (many of these strategies for code ‘evolution’, as well as others not mentioned here, are explained in greater detail by Cohen [4, pp. 199-215]):

- Members of the “Brain” strain of viruses are stealth floppy boot sector viruses.
- The “512” virus (also known as “Number of the Beast”) is a stealth file infecting virus.

**Encryption:** Encrypt the body of the virus not only with a variety of different keys but also with a variety of different encryption strategies (each requiring a different decryptor, of course). The encryption approach is a particularly common polymorphic trick — its wide use was facilitated by the distribution of a variety of object code modules (such as MtE and TPE) which could make any virus polymorphic. Additionally, this is the most straightforward polymorphic strategy which a virus might employ. One of the many viruses which employs a strategy of this sort is the “Tequila” virus [16].

**Instruction Equivalence:** Substitute for single instructions achieving equivalent effects in the virus code.

**Equivalent Instruction Sequences:** Replace one sequence of machine instructions with another which achieves the same final result.

**Instruction Reordering:** Sometimes it is possible to reorder a sequence of instructions in a variety of different ways and still achieve the same result. The “1260 virus” is one which alters the order of instructions in its decryption routine from infection to infection, and additionally inserts irrelevant instructions [16].

**Add or Remove Jumps/Calls:** Jumps or calls to blocks of code may be replaced by an instance of the block of code.

**Garbage Insertion:** For example, might add NOP instructions, or other instructions which do not otherwise achieve any useful purpose apart from obscuring the code’s functions. The “Tequila” virus employs a strategy of this sort [16].

**Build and Execute:** ‘Build’ instructions somewhere in memory (one byte or a sequence of bytes at a time) and then execute them. This obscures the instructions being constructed from observation until their execution.

**Tunnelling:** A technique sometimes used by viruses that attempts to bypass activity-monitoring virus detectors (for example, by bypassing operating system disk access functions and interpreting the contents of the disk itself, or calling the operating system’s functions directly to avoid any trap the activity monitor may have set in place) or otherwise subvert anti-virus techniques and strategies.
Cavity Virus: A variety of file infecting virus which overwrites a portion of the file which is filled with the same value, such as a region filled with zero bytes. Such a virus will not alter the total length of the file.

Antidebugger Mutations: Various tricks can be employed to make disassembling and tracing the program code difficult. For example, the program code might be arranged in such a way that an instruction or set of instructions is hidden from casual inspection, so that the function of the code is no longer readily apparent from its disassembly. The debugger itself could be manipulated in the course of tracing a program by tampering with the debugger’s address space.

The speed with which a virus reproduces and infects other files is also important. Some viruses spread particularly quickly. For example, a virus might be programmed to infect any executable file when the file is opened, for whatever reason. These are the so-called fast infectors.

A slow infector is a virus which only infects files as they are created, immediately prior to or following a legitimate change (that is, some change that the user wants to have happen), or as files are copied, perhaps onto a floppy disk [3, 9, 17] [4, p. 91].

7 Defenses Against and Detection For Viruses

There are a variety of defenses against viruses and ways of detecting their presence. A natural first question is: “Is it possible to detect all viruses?” Unfortunately not — it is mentioned in Cohen [4, pp. 64–68] that it is not possible to construct some program which correctly determines whether or not some other program is a virus. In fact, given a known virus, it isn’t even possible to systematically determine using a computer program if another program is infected with a virus derived from the original known virus in some way.

However, there are a variety of imperfect ways to detect the (possible) presence of computer viruses. Most forms of virus detection involve false positives, which occur when an object is identified as being infected by a virus when in fact it is clean, and false negatives, which occur when an object is passed as clean when in fact it has been infected with a virus. Ideally, false positives and false negatives will occur very infrequently.

Some techniques, such as scanning for viruses, often lead to a positive identification of a virus. In many cases, the infection (and some damage) caused by the virus is reversible.

Stealth techniques mean that attempts at virus detection that involve manipulating file objects on disk will not necessarily be effective if the virus is present and active in memory. For this reason, it is recommended that before attempting to detect a virus, the computer in question be rebooted from an uninfected, locked, floppy disk containing a clean version of whatever anti-virus software is needed to search for viruses.

Known-Virus Scanning: Attempts to identify viruses by scanning files for certain strings of bytes known to occur in particular viruses. Simple scanners which perform only such searches will fail to detect numerous polymorphic viruses, so many also employ some more advanced techniques (such as heuristic analysis, to detect suspicious code fragments, or algorithmic analysis, to detect complex polymorphic viruses). Scanners often have difficulty detecting new viruses, and they require frequent updating.

Heuristic Analysis: Attempts to identify a possible virus by looking for code that performs functions which in combination with each other are deemed to be suspicious. An example of a suspicious code fragment would be one that alters the first few bytes of an executable file in memory — this would be required so that an infected executable could run normally when infected with a virus.

Behaviour Blocker/Monitor: Attempts to detect viruses based on patterns of virus activity. This approach has problems because many of the actions performed by a virus are perfectly legitimate under other circumstances. These methods may be ineffective if a tunnelling virus is infecting the system — such viruses are frequently able to bypass the methods used by a monitor to detect virus activity.

Integrity Checker/Integrity Shell: Integrity checking involves collecting a database of signatures for each file which is likely to be the target of a virus infection (such as application programs). If at a later date this signature can be determined to have changed, then the file has been altered, possibly as a result of being infected by a virus. This method will not detect viruses before infection takes place. There are a variety of enhancements to the basic method outlined here which must be implemented (for example, a companion virus does not necessarily modify the item it “infects”. So integrity checkers must attempt to identify the presence of a companion virus by other means). An integrity shell [4, pp. 83–93] is a more sophisticated approach which involves
Checking every object on which some object X (which the user wishes to use in some way) depends.

An integrity checker is an example of a generic anti-virus program — it is not targeted at a specific virus or class of viruses and so will rarely need updating. Integrity checking issues are extensively discussed in papers by Bontchev [3] and Radai [9].

There are a variety of other ways to help prevent virus infections. For example, as the great majority of PC virus infections are boot sector viruses, changing the order in which the computer searches its disk drives for a bootable disk is an effective defense in many cases (an apparently common setting is to attempt to boot a floppy disk before attempting to boot the hard disk). Precautions will still have to be taken to deal with multipartite and other non-boot sector viruses, of course.

Cohen outlines a number of strategies that will prevent or hinder computer viruses spreading throughout a computer system or computer network [4, pp. 57-64]. The most interesting of these approaches is that of limited sharing. The best that can be done to limit sharing in a transitive information network that implements sharing is to base the structure on a "partially ordered set", or POset. Information can flow in only one direction, for example, from Host A to Host B but not from Host B to Host A. This effectively limits the possible range of a viral infection, and also helps to trace the origin of any suspected infection, as the source of the infection would be one of a limited number of machines which had access to all of the machines on which the infection was ultimately detected.

The use of a "vaccine" against certain computer viruses is a technique no longer widely practiced. Most viruses check a potential infection target to make sure it is not infected by that particular type of virus (to prevent multiple infections), so infection by a particular virus could be prevented by marking executables so that they appeared to be already infected (hence the name "vaccine"). The large number of viruses and the fact that some virus' identification techniques are mutually contradictory means that this technique is no longer workable.

8 Problems with and Attacks Against Anti-Virus Measures

Virus scanners need updating with great frequency because of the speed with which new viruses are created and released. They are popular for a number of reasons:

- A scanner is usually straightforward to use.
- When a virus is detected it can often be positively identified, and many scanners include facilities for "disinfecting" infected files.
- A scanner is the most reliable means of detecting a known virus in a new file; other techniques are not necessarily applicable (for example, an integrity checker cannot be used to check a newly-obtained program for viruses, because there is no way of determining what the signature of an uninfected program should be).

A scanner will often perform poorly when attempting to detect unknown viruses.

Polymorphism was at one time an effective attack against scanners which merely searched for strings of bytes known to characterise certain viruses. Cohen states that "until several years after the MtE was spreading in the world, no scanner was able to pick up over 95% of infections" [4, p. 73]. The situation has improved greatly in recent times — most good scanners are capable of detecting the majority of polymorphic viruses.

As mentioned in Section 7, stealth viruses may cause problems if the virus is present in memory when using a virus scanner or integrity checker, as files presented for inspection may appear clean when in fact they are not. Or a variety of fast infector stealth virus may infect many files opened for checking. This represents a serious cleanup problem in an environment with many executable files. So an attempt must be made to identify such viruses in memory.

Activity monitoring programs require updating as well, to cope with new virus behaviours. There are some varieties of virus behaviour which are not readily detectable by a monitor — such as infecting only files which are about to be modified in any case. One particular virus, the "Darth Vader" virus, was designed to avoid alerting an activity monitor program by attaching itself only to certain files as they were copied [3] — the executable object was modified while its code was present in memory awaiting being written to disk. This also presented an effective attack against integrity checkers at that time.

Activity monitors may also flag legitimate actions as suspicious, since the functions used by computer viruses have legitimate uses as well. They might also be bypassed by a tunnelling virus or disabled in memory [3, 9].

Slow infectors, infecting only when executables are being legitimately altered, are a concern for integrity checking software. The virus infection may go unnoticed, because the integrity checker doesn't have any signatures in its database with which to compare that of the (new) file. An example of a common legitimate change to an existing
executable file is the addition of patches to the file by an updater program. Radai [9] suggests that it would be possible to detect the presence of a slow virus by creating a series of small executable files in the hope that one will be infected by a slow virus. This will not establish which other newly-created or modified objects are infected with the slow virus, however.

Programs which cause modifications to their own executable code will also cause problems for integrity checkers.

Integrity checkers will not be effective against all types of viruses. As integrity checking is usually applied only to hard disks (its application to floppy disks is not practical, as the contents of a floppy disk are frequently modified) a virus which infects floppy disks only and ignores hard disks would go unnoticed [9, 3]. The “Brain” virus, an early DOS virus, is an example of a virus which ignores hard disks.

Finally, integrity checkers need to be carefully constructed so that the database of checksums is not easily compromised by a virus. For example, a virus might attempt to forge an entry in the integrity checker’s signature database for a program which is a target of infection.

9 Recent Work

9.1 Companion Viruses and the Macintosh

Macintosh viruses infect application files, the System file (a file present on every bootable Macintosh disk which holds many resources used by the operating system) or system extensions (small files containing executable code which load every time the Macintosh is started up and which add extra functionality to the operating system). A very few viruses which no longer work under recent releases of the operating system infected the Macintosh by more unusual means.

The Macintosh does not have any features which correspond to the preference of DOS to execute .COM files before .EXE files of the same name (when the selection is not made explicit). Nor does it feature the concept of a “path” along which the operating system searches for applications to execute if the desired application is not in the current directory, as DOS does. So a companion virus which does not actually alter the target application is not implementable in the same manner as under DOS.

However, it is possible to produce a virus with many of the same characteristics as a companion virus by manipulating a Macintosh disk’s Desktop Database. Each disk or disk partition will usually have its own Desktop Database, storing information about the application files on that disk or disk partition. This attack seems not to have been explored to date.

Macintosh files have both a file type (for example, ‘TEXT’ denotes a plain text file) and a creator. Each application should have a unique creator code, with which the files that the application creates are marked. The “Finder” (the part of the Macintosh operating system which is responsible for presenting the graphical user interface and managing user interactions) stores information about file creators which allows it to determine what application should be executed (“launched”) when a document icon is double-clicked, and what icon should be displayed for a file of a certain type and creator. This information is a part of the information stored by the Finder in a disk’s Desktop Database.

When there is more than one application present with the same creator, then the Finder launches the application with the most recent creation date when documents are double-clicked. Applications which are launched as the result of a user double-clicking a document icon are sent by the operating system what is referred to as a high level event or Apple Event, detailing which document or documents are to be manipulated.

A viral application, with a more recent creation date than the infected application, would be launched by the Finder before the target application, and would then have an opportunity to infect further applications (for example, any application currently running which has not already been infected, or perhaps by scanning the directory tree for uninfected applications, processing only a few directories at a time so as to avoid detection), or to perform other actions. The viral application would then pass control to the infected application. The Apple Event which details the documents to be processed can easily be passed on to the target application, so that there is little outward evidence that anything unusual has taken place.

This method of infection seems to function well on hard drives with single and multiple partitions, and should infect Macintosh file servers as well.

The ability of such a virus to conceal itself is limited, being restricted to setting the position of its document icon displayed by the Finder to some point off the edge of the display window. Such a defense is readily overcome with no special tools.

More work needs to be done on the passing of Apple Events by the viral application to the target application (although problems in this area are not a major handicap — an infected application may perform strangely when launched, but will subsequently behave normally). An Apple Event will sometimes require that a reply be returned by the receiving application. These replies can only
be generated by the infected application, so some means must be found for the viral application to pass the reply to the reply event's intended destination.

Like companion viruses of DOS, this attack avoids altering any existing executable code or system resources.

9.2 Macro Viruses

A traditional virus can be quite difficult to write, as there are many factors which must be considered if the virus is to work successfully and be able to avoid detection on a wide range of systems running a variety of software products.

Simple macro viruses, however, are not hampered by many of the same considerations. As an exercise, a number of very simple macro viruses were implemented by Horton. A first attempt required only a single macro, AutoClose. This attempt was very crude and would be easily detected by an alert user, as infecting a document when it was closed (causing the AutoClose macro to run) required that the document be saved to disk with the viral macro attached.

A later, slightly more sophisticated offering was implemented using two macros, AutoOpen and FileSaveAs. The AutoOpen macro would be activated whenever an infected document is opened in Word. If Word's Normal template is not already infected, the virus then infects the Normal template, which is a global macro file visible by all documents, by copying the viral macros to the template.

From this point on, whenever the user selects Save As... from the File menu, the FileSaveAs macro (now attached to the Normal template) is activated instead. It performs the same tasks as the usual menu option, with the exception that before the document is saved it is converted to a template to which the viral macros are then attached.

Neither of the two macro viruses created implemented any "payload" macros, although the addition of a simple payload would be trivial. The viruses are not remarkable for the techniques they employ, which are found in other macro viruses.

The first implementation attempt of a macro virus was accomplished in approximately eight hours by a computer science graduate previously unfamiliar with Microsoft Word and WordBasic. No documentation was available other than the "Help" system of Microsoft Word and a freely available description of some common macro viruses ([15]).

The second attempt required another hour, and was sufficiently sophisticated that it would be able to replicate unnoticed. A greater familiarity with WordBasic would be required to write anything more sophisticated, but the exercise illustrates the fact that macro viruses, because of their ease of construction, permit a much wider pool of computer users to write computer viruses in a very short space of time.

Macro viruses are quite widespread and so samples are often easy to obtain. Furthermore, understanding the code of a macro virus (assuming that the macros are not marked as ExecuteOnly) is very much less difficult than interpreting the assembly language in which non-macro viruses are typically implemented. The study of techniques used to implement other macro viruses is one straightforward method of improving virus writing.

More sophisticated macro viruses might target specific computer systems by attempting to "drop" a non-macro filesystem virus into the target system, or perform other operating system dependent actions. This would require greater familiarity with WordBasic or other macro languages than are demonstrated in a simple macro virus, such as the ones created as a part of this exercise.

10 Summary

This paper has presented a definition of computer viruses as the term is commonly used, as well as attempting to explain the meaning of some other terms, such as "worm", which are often closely associated with viruses. The various types of viruses that currently exist and various methods used by those viruses for infection have been explained, and the types and numbers of viruses likely to be found in the wild have been considered.

Some ways in which macro viruses operate within Microsoft Word 6 are discussed.

The various ways viruses currently use to hinder detection and delay analysis once they have been detected were discussed. Commonly available antivirus measures were explained, as well as faults and problems that these methods are known to have.

Finally, some recent work on the ease of implementation of simple macro viruses and the implementation of a companion virus-type attack for the Macintosh was discussed.

References


[3] V. Bontchev. Possible Virus Attacks Against Integrity Programs and how to prevent


Appendix B

Source code for Sequence Search
/*
 * Instructions for use:
 *
 * Compile this source file to generate an executable, run it to
 * eventually produce results. Program must be configured at
 * compile time.
 *
 * Compile-time configuration:
 *
 * The following symbols may be defined:
 *
 * N: an integer, the length of each sequence. Must be odd.
 * XN: an integer, the count of +ve and -ve X's.
 * YN: an integer, the count of +ve and -ve Y's.
 *
 * Note that only certain combinations of XN and YN are supported.
 * The following combinations have been used extensively:
 * (XN, YN): (1, 9)
 * (4, 9)
 * (1, 16)
 *
 * The code should compile on UNIX or Macintosh using most major
 * compilers.
 *
 * The compiler used must support the type 'unsigned long long'
 * to get 64-bit unsigned integers.
 */
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <limits.h>
#undef NDEBUG
#include <assert.h>

//#define _PRINT_COUNTS_

#ifdef macintosh
#ifdef _MWERKS_
#include <SIOUX.h>
#endif
#include <Sound.h>
#include <Speech.h>
#endif
#include <Gestalt.h>

#ifndef DISABLEJSPEECH
#endif

#ifdef Unix

#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
#endif

#define BILLION ((float)1000000000)

#ifndef N
#define N 15
#endif

#define HN ((N-1) / 2)

/*
 * Actual numbers are used to represent X's and Y's.
 * These numbers should be small and have no common
 * factors.
 */
#define X 2
#define Y 9

#ifndef XN
#define XN 1
#endif

#ifndef YN
#define YN 9
#endif

/*
 * 'unsigned long long' is a cumbersome type. Simplify
 * its name.
 */
typedef unsigned long long uLL;
typedef int tPSumArr[HN];

#define MAX_MB 15

/*
 * Number of precalculated partial sums that will fit into
 * a MAX_MB block of memory.
 */
/*

//#define MAX_SEQ  ((MAX_MB * 1024 * 1024) / sizeof(tPSumArr))
#define MAX_SEQ ((30 * 1024) / sizeof(tPSumArr))

#else define macintosh
#define TIMEFUNC() clock()
#define TIMER_RES Clocks.PER_SEC
#endif
#endif Unix
#if define TIMEFUNC() gethrvtime()
// #define TIMER_RES 1000000000
#define TIMEFUNC(MyUnixTimer())
#define TIMER_RES 1
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif

#define UNIX.Long.Long
#endif

int main( void );
int sym[] = {X, -X, Y, -Y, 0};
#endif
#else define UNIX.Long.Long
#endif

#if (defined(_MWERKS_) && (_MWERKS_ < 0x2100))
#define DODGY_MWERKS.Long.Long
#else
#define UNIX.Long.Long
#endif

int main( void );
int sym[] = {X, -X, Y, -Y, 0};
#if ((XN == 5) && (YN == 20))
// Symbol counts for (5, 20).

#define NUMBER_COUNTS 48

int SymCounts[][2][4] = {
    { { 1, 0, 11, 7 }, { 3, 1, 0, 2 } },
    { { 1, 0, 10, 6 }, { 3, 1, 1, 3 } },
    { { 1, 0, 9, 5 }, { 3, 1, 2, 4 } },
    { { 1, 0, 8, 4 }, { 3, 1, 3, 5 } },
    { { 1, 0, 7, 3 }, { 3, 1, 4, 6 } },
    { { 1, 0, 6, 2 }, { 3, 1, 5, 7 } },
    { { 1, 0, 5, 1 }, { 3, 1, 6, 8 } },
    { { 1, 0, 4, 0 }, { 3, 1, 7, 9 } },
    { { 2, 1, 11, 7 }, { 2, 0, 0, 2 } },
    { { 2, 1, 10, 6 }, { 2, 0, 1, 3 } },
    { { 2, 1, 9, 5 }, { 2, 0, 2, 4 } },
    { { 2, 1, 8, 4 }, { 2, 0, 3, 5 } },
    { { 2, 1, 7, 3 }, { 2, 0, 4, 6 } },
    { { 2, 1, 6, 2 }, { 2, 0, 5, 7 } },
    { { 2, 1, 5, 1 }, { 2, 0, 6, 8 } },
    { { 2, 1, 4, 0 }, { 2, 0, 7, 9 } },
    { { 0, 1, 11, 7 }, { 1, 3, 0, 2 } },
    { { 0, 1, 10, 6 }, { 1, 3, 1, 3 } },
    { { 0, 1, 9, 5 }, { 1, 3, 2, 4 } },
    { { 0, 1, 8, 4 }, { 1, 3, 3, 5 } },
    { { 0, 1, 7, 3 }, { 1, 3, 4, 6 } },
    { { 0, 1, 6, 2 }, { 1, 3, 5, 7 } },
    { { 0, 1, 5, 1 }, { 1, 3, 6, 8 } },
    { { 0, 1, 4, 0 }, { 1, 3, 7, 9 } },
    { { 1, 2, 11, 7 }, { 0, 2, 0, 2 } },
    { { 1, 2, 10, 6 }, { 0, 2, 1, 3 } },
    { { 1, 2, 9, 5 }, { 0, 2, 2, 4 } },
    { { 1, 2, 8, 4 }, { 0, 2, 3, 5 } },
    { { 1, 2, 7, 3 }, { 0, 2, 4, 6 } },
    { { 1, 2, 6, 2 }, { 0, 2, 5, 7 } },
    { { 1, 2, 5, 1 }, { 0, 2, 6, 8 } },
    { { 1, 2, 4, 0 }, { 0, 2, 7, 9 } },
    { { 1, 0, 7, 11 }, { 3, 1, 2, 0 } },
    { { 1, 0, 6, 10 }, { 3, 1, 3, 1 } },
    { { 1, 0, 5, 9 }, { 3, 1, 4, 2 } },
    { { 1, 0, 4, 8 }, { 3, 1, 5, 3 } },}
# if ((XN == 9) && (YN == 9))
// Symbol counts for (9, 9).

#define NUMBER_COUNTS 16

int SymCounts[][2][4] = {
    {{6, 3, 0, 0}, {0, 0, 6, 3}},
    {{6, 3, 1, 1}, {0, 0, 5, 2}},
    {{6, 3, 2, 2}, {0, 0, 4, 1}},
    {{6, 3, 3, 3}, {0, 0, 3, 0}},
    {{5, 2, 0, 0}, {1, 1, 6, 3}},
    {{5, 2, 1, 1}, {1, 1, 5, 2}},
    {{5, 2, 2, 2}, {1, 1, 4, 1}},
    {{5, 2, 3, 3}, {1, 1, 3, 0}},
    {{4, 1, 0, 0}, {2, 2, 6, 3}},
    {{4, 1, 1, 1}, {2, 2, 5, 2}},
    {{4, 1, 2, 2}, {2, 2, 4, 1}},
    {{4, 1, 3, 3}, {2, 2, 3, 0}},
    {{3, 0, 0, 0}, {3, 3, 6, 3}},
    {{3, 0, 1, 1}, {3, 3, 5, 2}},
    {{3, 0, 2, 2}, {3, 3, 4, 1}},
    {{3, 0, 3, 3}, {3, 3, 3, 0}}
};
#endif

#if ((XN == 4) && (YN == 9))
// Symbol counts for (4, 9).

// #define NUMBER_COUNTS 16
// Ignoring the negation of the B-sequence.
#define NUMBER_COUNTS 8

int SymCounts[][2][4] = {
    {{3, 1, 0, 0}, {0, 0, 6, 3}},
    {{3, 1, 1, 1}, {0, 0, 5, 2}},
    {{3, 1, 2, 2}, {0, 0, 4, 1}},
    {{3, 1, 3, 3}, {0, 0, 3, 0}},
    {{2, 0, 0, 0}, {1, 1, 6, 3}},
    {{2, 0, 1, 1}, {1, 1, 5, 2}},
    {{2, 0, 2, 2}, {1, 1, 4, 1}},
    {{2, 0, 3, 3}, {1, 1, 3, 0}},
    {{3, 1, 0, 0}, {0, 0, 3, 6}},
    {{3, 1, 1, 1}, {0, 0, 2, 5}},
    {{3, 1, 2, 2}, {0, 0, 1, 4}},
    {{3, 1, 3, 3}, {0, 0, 0, 3}},
    {{2, 0, 0, 0}, {1, 1, 3, 6}},
    {{2, 0, 1, 1}, {1, 1, 2, 5}},
    {{2, 0, 2, 2}, {1, 1, 1, 4}},
    {{2, 0, 3, 3}, {1, 1, 0, 3}},
};

#endif

#if ((XN == 1) && (YN == 9))
// Symbol counts for (1, 9).
#endif

#if ((XN == 1) && (YN == 9))
// Symbol counts for (1, 9).
#endif

/*
 * OK: The only difference between these sequence counts and the
 * previous set is that the counts of symbols in the B-sequence
 * are swapped for -ve and +ve symbols.
 * But consider how the sum [Sum]BiBi-j works:
 *  -ve * -ve = +ve
 *  +ve * +ve = +ve
 *  -ve * +ve = -ve
 *  +ve * -ve = -ve
 * I get -ve's when I multiply symbols of dissimilar sign. But I
 * get exactly the same -ve's if the sign of each symbol in the
 * sequence was reversed anyway!
 */

{ { 3, 1, 0, 0 }, { 0, 0, 6, 3 } },
{ { 3, 1, 1, 1 }, { 0, 0, 5, 2 } },
{ { 3, 1, 2, 2 }, { 0, 0, 4, 1 } },
{ { 3, 1, 3, 3 }, { 0, 0, 3, 0 } },
{ { 2, 0, 0, 0 }, { 1, 1, 6, 3 } },
{ { 2, 0, 1, 1 }, { 1, 1, 5, 2 } },
{ { 2, 0, 2, 2 }, { 1, 1, 4, 1 } },
{ { 2, 0, 3, 3 }, { 1, 1, 3, 0 } },
};
#endif
#define NUMBER_COUNTS 4

int SymCounts[][2][4] = {
    {{1,0, 1,1}, {0,0, 5, 2 }},
    {{1,0, 3,3 }, {0,0, 3, 0 }},
    {{1,0, 2,2 }, {0,0, 4, 1 }},
    {{1,0, 0,0 }, {0,0, 6, 3 }}
};
#endif

#if ((XN == 1) && (YN == 16))
// Symbolic counts for (1, 16).
#endif

#define NUMBER_COUNTS 7

int SymCounts[][2][4] = {
    {{1,0, 0,0 }, {0,0, 10, 6 }},
    {{1,0, 1,1 }, {0,0, 9, 5 }},
    {{1,0, 2,2 }, {0,0, 8, 4 }},
    {{1,0, 3,3 }, {0,0, 7, 3 }},
    {{1,0, 4,4 }, {0,0, 6, 2 }},
    {{1,0, 5,5 }, {0,0, 5, 1 }},
    {{1,0, 6,6 }, {0,0, 4, 0 }}
};
#endif

/*
 * How many symbols in each sequence for each class of
 * sequences to be tested?
 */
int SymSums[NUMBER_COUNTS][2];

#ifdef macintosh
static int gHasSpeech = 0;
static Str255 gSpeechString;

static void HasSpeechMgr( void ) {
    long feature;
    OSErr err;

    err = Gestalt(gestaltSpeechAttr, &feature);
    if (err == noErr)
        gHasSpeech = feature & (1 << gestaltSpeechMgrPresent);
    else
        gHasSpeech = 0;
}
static void MySpeakString( Str255 string ) {
    short activeChannels;

    if (gHasSpeech) {
        activeChannels = SpeechBusy();
        SpeakString(string);
        while (SpeechBusy() != activeChannels)
            ;
    }
}

static void MySpeakNumber( long number ) {
    if (gHasSpeech) {
        NumToString(number, gSpeechString);
        MySpeakString(gSpeechString);
    }
}

#endif

static void PrintSequence( int sequence[] ) {
    int i;

    for (i = 0; i < N; ++i) {
        switch (sym[sequence[i]]) {
        case X:
            printf("%3s", "X");
            break;
        case -X:
            printf("%3s", "-X");
            break;
        case Y:
            printf("%3s", "Y");
            break;
        case -Y:
            printf("%3s", "-Y");
            break;
        case 0:
            printf("%3s", "0");
            break;
        }
    }
    printf("\n");
}
static int NextSequence(
    int MaxSymCounts[],
    int MaxSyms,
    int Counts[],
    int sequence[],
    int pos)
{
    register int i, lastval, countsum, bFinding;

    /*
     * If the last non-zero element in the sequence is not the
     * last element, shift it along.
     */
    if (pos < N-1) {
        // sequence[pos+1] = sequence[pos];
        // sequence[pos++] = 4;
        if (pos != 0) {
            sequence[pos+1] = sequence[pos];
            sequence[pos++] = 4;
        }
    } else {
        --(Counts[sequence[0]]);
        sequence[0] = 4;
        pos = -1;
    }
    }
else {
    --(Counts[sequence[pos]]);
    sequence[pos] = 4;
    /*
    while ((pos >= 0) && (sequence[pos] == 4))
    -pos;
    */
    do
    --pos;
    while ((pos >= 0) && (sequence[pos] == 4));
    if (pos ≥ 0) {
        countsum = MaxSyms - 1;
        do {
            bFinding = 0;
            /*
             * Advance a symbol from one non-zero symbol to
             * another non-zero symbol if possible, to zero if not.
             */
        }
lastval = sequence[pos];
--(Counts[lastval]);
--countsum;
sequence[pos] = 4;
if (pos == 0)
    return -1;
for (i = lastval+1; i < 4; ++i) {
    if (Counts[i] ≠ MaxSymCounts[i]) {
        ++(Counts[i]);
        sequence[pos] = i;
        break;
    }
}
if ((i == 4) && ((pos + MaxSyms - countsum) ≥ N)) {
    while ((pos ≥ 0) && (sequence[pos] == 4))
        --pos;
    do
        --pos;
    while ((pos ≥ 0) && (sequence[pos] == 4));
    if (pos ≥ 0)
        bFinding = 1;
}
while (bFinding);
/*
* Fill in the rest of the sequence with the remaining symbols.
*/
if (pos ≥ 0) {
    for (i = 0; i < 4; ++i) {
        while (Counts[i] < MaxSymCounts[i]) {
            sequence[++pos] = i;
            ++(Counts[i]);
        }
    }
}
assert(pos < N);
return pos;
}

static int InitSequence(
    int MaxSymCounts[],
int Counts[],
int sequence[] ) :
{
    int i, pos, min;

    min = INT_MAX;
    /*
     * Set the first (fixed) symbol in the sequence to be the
     * one that occurs the fewest times greater than zero, as
     * this will greatly reduce the number of rotationally
     * equivalent sequences that must be considered.
     */
    for (i = 0; i < 4; ++i) {
        if ((MaxSymCounts[i] > 0) && (MaxSymCounts[i] < min)) {
            min = MaxSymCounts[i];
            pos = i;
        }
    }
    sequence[0] = pos;
    ++(Counts[pos]);
    pos = 0;
    for (i = 0; i < 4; ++i) {
        while (Counts[i] < MaxSymCounts[i]) {
            sequence[++pos] = i;
            ++(Counts[i]);
        }
    }
    assert((pos < N) && (pos > 0));
    return pos;
}

static void PrintSequenceUsingIndex(
    unsigned long search_idx,
    int MaxSymCounts[],
    int MaxSyms )
{
    int i, sequence[N], pos, Counts[4], index;

    memset(Counts, 0, sizeof(Counts));
    for (i = 0; i < N; ++i)
        sequence[i] = 4;
    index = 0;
    pos = InitSequence(MaxSymCounts, Counts, sequence);
    while (search_idx != index) {
        pos = NextSequence( MaxSymCounts, MaxSyms, Counts, sequence, pos );
    }
static void EvaluateSums(int sequence[], int sums_of_products[]) {
    register int i, j, l, sum_of_products;
    for (i = 0; i < HN; ++i) {
        sum_of_products = 0;
        l = i;
        for (j = 0; j < N; ++j) {
            ++l;
            // MUCH better than 'l -= (l >= N) * N'
            if (l >= N)
                l = N;
            sum_of_products += sym[sequence[j]] * sym[sequence[l]];
        }
        sums_of_products[i] = sum_of_products;
        // assert((sum_of_products < (1 << 29)) && (sum_of_products > -
        (1 << 29)));
    }
}

static void PrintSymbolCounts(int SymbolCounts[]) {
    int i;
    printf(" ( ");
    for (i = 0; i < 4; ++i) {
        printf(" %d", SymbolCounts[i]);
        if (i != 3)
            printf(" , ");
    }
    printf(" ) ");
}

static void CountSequences(int pair, uLL *CountA, uLL *CountB) {
    int i,
        sequence[N],
        pos,
        Counts[4];
    uLL sequence_count;
    if ((SymSums[pair][0] > N) || (SymSums[pair][1] > N)) {
        // Should never happen, but be sensible if it does.
*CountA = *CountB = 0;

memset(Counts, 0, sizeof(Counts));
for (i = 0; i < N; ++i)
    sequence[i] = 4;
for (i = 0; i < 2; ++i) {
    sequence_count = 0;
    pos = InitSequence(SymCounts[pair][i], Counts, sequence);
    while (pos > 0) {
        ++sequence_count;
        pos = NextSequence(SymCounts[pair][i], SymSums[pair][i], Counts,
                            sequence, pos);
    }
    if (i == 0)
        *CountA = sequence_count;
    else
        *CountB = sequence_count;
}

static unsigned long AllocatePSumArray(
    uLL total_sequences,
    tPSumArr **prodsums)
{
    unsigned long bytes_to_allocate, sequences_per_chunk, chunks;

    if (total_sequences > MAX_SEQ) {
        chunks = (total_sequences / MAX_SEQ) + (((total_sequences % MAX_SEQ) != 0) ? 1 : 0);
        sequences_per_chunk = MAX_SEQ;
    } else {
        sequences_per_chunk = total_sequences;
        chunks = 1;
    }
    bytes_to_allocate = sizeof(tPSumArr) * sequences_per_chunk;
    *prodsums = (tPSumArr *)malloc(bytes_to_allocate);
    if (*prodsums == NULL) {
        fprintf(stderr, "Damn! Couldn't allocate the required memory.\n");
        exit(1);
    }
    printf("%lu bytes allocated; %lu chunk(s).\n", bytes_to_allocate, chunks);
    return sequences_per_chunk;
```c
static int ComputePartialSums(
    unsigned long sequences_per_chunk,
    tPSumArr *prodsums,
    int sequence[],
    int *posPtr,
    int Counts[],
    int MaxSymCounts[],
    int MaxSyms)
{
    int i;
    unsigned long sequence_count;

    sequence_count = 0;
    if (prodsums != NULL) {
        if (*posPtr < 0) {
            // Initialise the sequence if (pos < 0) when function called.
            memset(Counts, 0, (sizeof(int) * 4));
            for (i = 0; i < N; ++i)
                sequence[i] = 4;
            *posPtr = InitSequence(MaxSymCounts, Counts, sequence);
        }
        while ((*posPtr >= 0) && (sequence_count != sequences_per_chunk)) {
            EvaluateSums(sequence, prodsums[sequence_count]);
            ++sequence_count;
            *posPtr = NextSequence(MaxSymCounts, MaxSyms, Counts, sequence,
                                    *posPtr);
        }
    }
    return sequence_count;
}

static tPSumArr *ComputeAllSums(
    unsigned long total_sequences,
    int MaxSymCounts[],
    int MaxSyms)
{
    int i,
        sequence[N],
        pos,
        Counts[4];
    tPSumArr *prodsums;
    unsigned long bytes;
```
bytes = sizeof(tPSumArr) * total_sequences;
if (bytes > (20 * 1024 * 1024)) {
    fprintf(stderr, "Damn! Would need to allocate more than 20 MB here!
    ");
    exit(1);
}
prodsums = (tPSumArr *)malloc(bytes);
if (prodsums ^ NULL) {
    printf("%lu bytes allocated.\n", bytes);
    memset(Counts, 0, sizeof(Counts));
    for (i = 0; i < N; ++i)
        sequence[i] = 4;
    i = 0;
    pos = InitSequence( MaxSymCounts, Counts, sequence );
    while (pos >= 0) {
        EvaluateSums( sequence, prodsums[i] );
        ++i;
        pos = NextSequence( MaxSymCounts, MaxSyms, Counts, sequence, pos );
    }
    return prodsums;
}

#ifndef DODGY_JVWERKS_LONG_LONG
if (hours ^ 0) printf("%Lu hours ", hours);
if (minutes != 0)
    printf("%Lu minutes ", minutes);
if ((seconds != 0) || ((hours == 0) && (minutes == 0)))
    printf("%Lu seconds ", seconds);
#endif
#else
    if (hours != 0)
        printf("%lld hours ", hours);
    if (minutes != 0)
        printf("%lld minutes ", minutes);
    if ((seconds != 0) || ((hours == 0) && (minutes == 0)))
        printf("%lld seconds ", seconds);
#endif
}

static void CheckSequences( void ) {
    int i, j, l, m, which_seq, sum_of_products, sums_of_productsBS[HN];
    int bSequencePrinted;

    unsigned long mincount,
        sequenceA_idx,
        sequences_per_chunk,
        found_sequence_count;

    uLL sequence_counts[2],
        sequences_examined;

    uLL total_loops, loops;

    int CountsA[4],
        posA,
        CountsB[4],
        posB,
        sequenceA[N],
        sequenceB[N];

    tPSumArr * prodsums;

#ifdef macintosh
    clock_t whole_start, start, stop;
#endif
#endif
#ifdef Unix
    time_t whole_start, start, stop;
#endif
int *tmp;

found_sequence_count = 0;
sequences_examined = 0;
bSequencePrinted = 0;
whole_start = TIMEFUNCQ;
fflush(stdout);
total_loops = 0;
for (m = 0; m < NUMBER_COUNTS; ++m) {
    ifdef macintosh
        MySpeakNumber(m+1);
    endif
    printf("Processing %d of %d ... ", (m+1), NUMBER_COUNTS);
    PrintSymbolCounts(SymCounts[m][0]);
    PrintSymbolCounts(SymCounts[m][1]);
    /*
     * Only process a particular combination of symbols when
     * the total number of symbols in each sequence is less
     * than or equal to the size of the sequence.
     */
    if ((SymSums[m][0] > N) || (SymSums[m][1] > N)) {
        printf("Skipped ... too many symbols.\n");
        continue;
    }
    CountSequences(m, &(sequence_counts[0]), &(sequence_counts[1]));
    ifdef DODGY_MWERKS_LONG_LONG
        printf(" (%Lu x %Lu = %.2f billion)\n", sequence_counts[0],
        sequence_counts[1], (sequence_counts[0] * sequence_counts[1]) / BILLION);
    endif
    ifdef UNIX_LONG_LONG
        printf(" (%llu x %llu = %.2f billion)\n", sequence_counts[0],
        sequence_counts[1], (sequence_counts[0] * sequence_counts[1]) / BILLION);
    endif
    fflush(stdout);
    start = TIMEFUNCQ;
    // Clear out the old sequence.
    for (j = 0; j < N; ++j)
        sequenceB[j] = 4;
    // Clear initial counts of symbols.
    memset(CountsB, 0, sizeof(CountsB));
posA = -1;
    loops = 0;
    sequenceA_idx = 0;
    if (sequence_counts[0] > sequence_counts[1]) {
        /*
* There are fewer possible sequences for the second of
* the two variable counts that make up this pair. Allocate
* a chunk of memory to hold the precomputations for the
* second, and iterate over the first.
*/
sequences_per_chunk = AllocatePSumArray(sequence_counts[1], &prodsums);
which_seq = 0;
}
else {
sequences_per_chunk = AllocatePSumArray(sequence_counts[0], &prodsums);
which_seq = 1;
}
do {
mincount = ComputePartialSums(sequences_per_chunk, prodsums,
sequenceA, &posA, CountsA, SymCounts[m][1-which_seq], SymSums[m][1-which_seq]);
posB = InitSequence(SymCounts[m][which_seq], CountsB, sequenceB);
if (mincount == 1) {
tmp = prodsums[0];
while (posB >= 0) {
for (i = 0; i < HN; ++i) {
++loops;
sum_of_products = 0;
l = i;
for (j = 0; j < N; ++j) {
++l;
if (l >= N)
1 -= N;
sum_of_products += sym[sequenceB[j]] * sym[sequenceB[l]];
}
if (tmp[i] + sum_of_products != 0)
break;
}
if (i == HN) {
if (!bSequencePrinted) {
PrintSequence(sequenceB);
PrintSequenceUsingIndex(sequenceA_idx,
SymCounts[m][1-which_seq], SymSums[m][1-which_seq]);
bSequencePrinted = 1;
}
++found_sequence_count;
goto done;
}
++sequences_examined;
posB = NextSequence(SymCounts[m][which_seq],
SymCounts[m][1-which_seq], SymSums[m][1-which_seq]);
else {
    while (posB > 0) {
        /*
        * Evaluate the sums of products for the current
        * sequence.
        */
        EvaluateSums(sequenceB, sums_of_productsBS);
        /*
        * Add sums of products for current sequence to
        * precalculated sums of products for the other
        * sequence.
        */
        for (i = 0; i < mincount; ++i) {
            tmp = prodsums[i];
            for (j = 0; j < HN; ++j) {
                ++loops;
                if (((tmp[j] + sums_of_productsBS[j]) != 0) { 
                    break;
                }
            }
            if (j == HN) {
                if (!bSequencePrinted) {
                    PrintSequence(sequenceB);
                    PrintSequenceUsingIndex((sequenceA_idx + i),
                    SymCounts[m][1-which_seq], SymSums[m][1-which_seq]);
                    bSequencePrinted = 1;
                }
                ++found_sequence_count;
                goto done;
            }
            ++sequences_examined;
        }
        posB = NextSequence( SymCounts[m][which_seq],
        SymSums[m][which_seq], CountsB, sequenceB, posB );
    }
    sequenceA_idx += mincount;
}
while (posA > 0);
free(prodsums);
prodsums = NULL;
total_loops += loops;
#ifdef DODGY_MWERKS_LONG_LONG
    printf("This: %Lu (%.2f billion) Total: %Lu (%.2f billion)\n", loops, (loops / BILLION), total_loops, (total_loops / BILLION));
#endif
#if defined UNIX_LONG_LONG
    printf("This: %llu (%.2f billion) Total: %llu (%.2f billion)\n", loops, (loops / BILLION), total_loops, (total_loops / BILLION));
#endif
bSequencePrinted = 0;
stop = TIMEFUNC();
printf("Finished %d of %d: ",(m+1), NUMBER_COUNTS);
PrintTime((stop - start) / TIMER_RES);
printf("\n");
#endif
#else
    printf("Sequences examined so far: %Lu (%.2f billion)\n", sequences_examined, (sequences_examined / BILLION));
#endif
#else
    printf("Sequences examined so far: %llu (%.2f billion)\n", sequences_examined, (sequences_examined / BILLION));
#endif
printf("\n");
fflush(stdout);
}
done:
if (prodsums) free(prodsums);
stop = TIMEFUNC();
printf("\n");
printf("Total sequences found: %lu\n", found_sequence_count);
#endif
#else
    printf("Total sequences examined: %Lu (%.2f billion)\n", sequences_examined, (sequences_examined / BILLION));
#endif
#else
    printf("Total sequences examined: %llu (%.2f billion)\n", sequences_examined, (sequences_examined / BILLION));
#endif
printf("Time taken: ");
PrintTime((stop - whole_start) / TIMER_RES);
printf("\n");
return;
}
* Printout a table of counts of sequences per symbol arrangement
* for sequences A and B, the total number of sequences for each
* pair of arrangements, and the total number of sequences across
* all pairs.
*/

```c
static void
PrintCountSequences( void ) {
    int i, j;

    uLL sequence_count, total_sequence_count;
    uLL sequence_counts[2];

    total_sequence_count = 0;
    for (i = 0; i < NUMBER_COUNTS; ++i) {
        /*
         * Skip it if there are too many symbols per sequence for
         * either sequence in the pair.
         */
        if ((SymSums[i][0] > N) || (SymSums[i][1] > N)) {
            PrintSymbolCounts(SymCounts[i][0]);
            PrintSymbolCounts(SymCounts[i][1]);
            printf("Skipped ... too many symbols per sequence for this N.\n");
            continue;
        }
        CountSequences(i, &(sequence_counts[0]), &(sequence_counts[1]));
        for (j = 0; j < 2; ++j) {
            PrintSymbolCounts(SymCounts[i][j]);
            if (DODGY_MWERKS_LONG_LONG)
                printf(" % -15Lu ", sequence_counts[j]);
            elseif UNIX_LONG_LONG
                printf(" % -15llu ", sequence_counts[j]);
            endif
            fflush(stdout);
        }
        sequence_count = sequence_counts[0] * sequence_counts[1];
        if (DODGY_MWERKS_LONG_LONG)
            printf(" = %Lu (%.2f billion)\n", sequence_count, (sequence_count / BILLION));
        elseif UNIX_LONG_LONG
            printf(" = %llu (%.2f billion)\n", sequence_count, (sequence_count / BILLION));
        endif
        total_sequence_count += sequence_count;
    }
}
```
```c
} #ifdef DODGY_MWERKS_LONG_LONG
    printf("\nTotal sequences = %Lu (%.2f billion)\n", 
           total_sequence_count, (total_sequence_count / BILLION)); #endif #ifdef UNIX_LONG_LONG
    printf("\nTotal sequences = %llu (%.2f billion)\n", 
           total_sequence_count, (total_sequence_count / BILLION)); #endif
#endif
printf("\n");
}

int main( void ) {
    int i, j, k;
    #ifdef _MWERKS_
        SIouxSettings.columns = 120;
    #endif
    printf("Start sequence finding ... (%d, %d), N = %d
(%s)\n", XN, YN, N, ID_STRING);
    fflush(stdout);
#if defined(macintosh) && !defined(DISABLE_SPEECH)
    HasSpeechMgr();
#endif
    for (i = 0; i < NUMBER_COUNTS; ++i) {
        for (j = 0; j < 2; ++j) {
            SymSums[i][j] = 0;
            for (k = 0; k < 4; ++k)
                SymSums[i][j] += SymCounts[i][j][k];
        }
        if ((SymSums[i][0] + SymSums[i][1]) + (XN + YN)) {
            fprintf(stderr, "Damn! Tabulated symbol counts are
incorrect!\n");
            exit(1);
        }
    }
#else
    PrintCountSequences();
#else
    CheckSequences();
#endif
    printf("Stop sequence finding.\n"); #ifdef macintosh
    SysBeep(50);
#endif
```
return 0;
}

Appendix C

Complete tables of Hadamard matrices derived from Williamson matrices for orders 12 through 148
<table>
<thead>
<tr>
<th>$t$</th>
<th>$n$</th>
<th>$\mu_1^2 + \mu_2^2 + \mu_3^2 + \mu_4^2$</th>
<th>$\mu_1$</th>
<th>$\mu_2$</th>
<th>$\mu_3$</th>
<th>$\mu_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>12</td>
<td>$1^2 + 1^2 + 1^2 + 3^2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>$1^2 + 1^2 + 3^2 + 3^2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>28</td>
<td>$1^2 + 1^2 + 1^2 + 5^2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>28</td>
<td>$1^2 + 3^2 + 3^2 + 3^2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>36</td>
<td>$1^2 + 1^2 + 3^2 + 5^2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>36</td>
<td>$3^2 + 3^2 + 3^2 + 3^2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>44</td>
<td>$1^2 + 3^2 + 3^2 + 5^2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>52</td>
<td>$1^2 + 1^2 + 1^2 + 7^2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>52</td>
<td>$1^2 + 1^2 + 5^2 + 5^2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>52</td>
<td>$3^2 + 3^2 + 3^2 + 5^2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>60</td>
<td>$1^2 + 1^2 + 3^2 + 7^2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>60</td>
<td>$1^2 + 3^2 + 5^2 + 5^2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>68</td>
<td>$1^2 + 3^2 + 3^2 + 7^2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>68</td>
<td>$3^2 + 3^2 + 5^2 + 5^2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table C.1: Hadamard matrices of orders 12, 20, 28, 36, 44, 52, 60 and 68 from Williamson Matrices
<table>
<thead>
<tr>
<th>$t$</th>
<th>$n$</th>
<th>$\mu_1^2 + \mu_2^2 + \mu_3^2 + \mu_4^2$</th>
<th>$N$?</th>
<th>$\mu_1$</th>
<th>$\mu_2$</th>
<th>$\mu_3$</th>
<th>$\mu_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>76</td>
<td>$1^2 + 1^2 + 5^2 + 7^2$</td>
<td></td>
<td>1</td>
<td>1</td>
<td>$1 + 2\omega_1 + 2\omega_8 - 2\omega_3$</td>
<td>$1 + 2\omega_2 + 2\omega_6 - 2\omega_4$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$-2\omega_5 - 2\omega_7 - 2\omega_9$</td>
<td>$1 + 2\omega_7 - 2\omega_3 - 2\omega_6$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$-2\omega_9$</td>
<td>$1 - 2\omega_1 - 2\omega_5$</td>
</tr>
<tr>
<td>19</td>
<td>76</td>
<td>$1^2 + 5^2 + 5^2 + 5^2$</td>
<td></td>
<td>1</td>
<td>$1 + 2\omega_9 - 2\omega_3$</td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 + 2\omega_9 + 2\omega_3 - 2\omega_6$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 - 2\omega_3 + 2\omega_6 - 2\omega_9$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 + 2\omega_4 + 2\omega_7 - 2\omega_5$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 + 2\omega_1 + 2\omega_6 - 2\omega_2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 + 2\omega_1$</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>76</td>
<td>$3^2 + 3^2 + 3^2 + 7^2$</td>
<td>No solutions.</td>
<td>1</td>
<td>$1 + 2\omega_9 - 2\omega_3$</td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 + 2\omega_9 + 2\omega_3 - 2\omega_6$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_9 - 2\omega_3$</td>
<td>$1 - 2\omega_3 + 2\omega_6 - 2\omega_9$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_9 - 2\omega_3$</td>
<td>$1 + 2\omega_4 + 2\omega_7 - 2\omega_5$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_9 - 2\omega_3$</td>
<td>$1 + 2\omega_1 + 2\omega_6 - 2\omega_2$</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>84</td>
<td>$1^2 + 1^2 + 1^2 + 9^2$</td>
<td></td>
<td>1</td>
<td>$1 + 2\omega_9 - 2\omega_3$</td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 + 2\omega_9 + 2\omega_3 - 2\omega_6$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 - 2\omega_3 + 2\omega_6 - 2\omega_9$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 + 2\omega_4 + 2\omega_7 - 2\omega_5$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 + 2\omega_1 + 2\omega_6 - 2\omega_2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 + 2\omega_1$</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>84</td>
<td>$1^2 + 3^2 + 5^2 + 7^2$</td>
<td></td>
<td>1</td>
<td>$1 + 2\omega_9 - 2\omega_3$</td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 + 2\omega_9 + 2\omega_3 - 2\omega_6$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 - 2\omega_3 + 2\omega_6 - 2\omega_9$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 + 2\omega_4 + 2\omega_7 - 2\omega_5$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 + 2\omega_1 + 2\omega_6 - 2\omega_2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 + 2\omega_1$</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>84</td>
<td>$3^2 + 5^2 + 5^2 + 5^2$</td>
<td></td>
<td>1</td>
<td>$1 + 2\omega_9 - 2\omega_3$</td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 + 2\omega_9 + 2\omega_3 - 2\omega_6$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 - 2\omega_3 + 2\omega_6 - 2\omega_9$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 + 2\omega_4 + 2\omega_7 - 2\omega_5$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 + 2\omega_1 + 2\omega_6 - 2\omega_2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_8 - 2\omega_1 + 2\omega_4 - 2\omega_2$</td>
<td>$1 + 2\omega_1$</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>92</td>
<td>$1^2 + 1^2 + 3^2 + 9^2$</td>
<td></td>
<td>1</td>
<td>$1 + 2\omega_9 + 2\omega_{11} - 2\omega_4$</td>
<td>$1 + 2\omega_5 - 2\omega_7$</td>
<td>$1 + 2\omega_1 + 2\omega_3 - 2\omega_{10}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_9 + 2\omega_{11} - 2\omega_4$</td>
<td>$1 + 2\omega_5 - 2\omega_7$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_9 + 2\omega_{11} - 2\omega_4$</td>
<td>$1 + 2\omega_1 + 2\omega_3 - 2\omega_{10}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$1 + 2\omega_9 + 2\omega_{11} - 2\omega_4$</td>
<td>$1 + 2\omega_2 + 2\omega_6$</td>
<td></td>
</tr>
</tbody>
</table>

Table C.2: Hadamard matrices of orders 76, 84 and 92 from Williamson Matrices
<table>
<thead>
<tr>
<th>$t$</th>
<th>$n$</th>
<th>$\mu_1 + \mu_2^2 + \mu_3^2 + \mu_4^2$</th>
<th>$N?$</th>
<th>$\mu_1$</th>
<th>$\mu_2$</th>
<th>$\mu_3$</th>
<th>$\mu_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>100</td>
<td>$1^2 + 1^2 + 7^2 + 7^2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$1 + 2\omega_6 + 2\omega_{12} - 2\omega_2$</td>
<td>$1 + 2\omega_8 + 2\omega_9 - 2\omega_1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>$1 + 2\omega_3 - 2\omega_9$</td>
<td>$1 + 2\omega_4 - 2\omega_{12}$</td>
<td>$-2\omega_3 - 2\omega_5 - 2\omega_7$</td>
<td>$-2\omega_4 - 2\omega_{10} - 2\omega_{11}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>$1 + 2\omega_3 - 2\omega_7$</td>
<td>$1 + 2\omega_4 - 2\omega_1$</td>
<td>$1 - 2\omega_1 - 2\omega_7$</td>
<td>$1 + 2\omega_6 + 2\omega_8 - 2\omega_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>$1 + 2\omega_6 - 2\omega_{11}$</td>
<td>$1 + 2\omega_3 - 2\omega_1 - 2\omega_{12}$</td>
<td>$1 + 2\omega_8 - 2\omega_9 - 2\omega_{10}$</td>
<td>$-2\omega_{11}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>$1 + 2\omega_7 + 2\omega_{10} - 2\omega_1$</td>
<td>$1 - 2\omega_5$</td>
<td>$1 + 2\omega_9 + 2\omega_{11} - 2\omega_3$</td>
<td>$1 + 2\omega_6 + 2\omega_{12}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>$1 + 2\omega_5 - 2\omega_{10}$</td>
<td>$1 + 2\omega_6 + 2\omega_{11} - 2\omega_2$</td>
<td>$1 + 2\omega_4 + 2\omega_9 + 2\omega_{12}$</td>
<td>$-2\omega_7 - 2\omega_8$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>$1 + 2\omega_1 + 2\omega_2 - 2\omega_3$</td>
<td>$1 + 2\omega_6 + 2\omega_9 - 2\omega_{10}$</td>
<td>$1 + 2\omega_7 + 2\omega_{11} - 2\omega_4$</td>
<td>$1 + 2\omega_8 + 2\omega_{12} - 2\omega_5$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>$1 + 2\omega_1 + 2\omega_9 - 2\omega_6$</td>
<td>$1 + 2\omega_7 + 2\omega_{12} - 2\omega_8$</td>
<td>$1 + 2\omega_2 + 2\omega_5 - 2\omega_{14}$</td>
<td>$1 + 2\omega_{10} + 2\omega_{11} - 2\omega_3$</td>
</tr>
<tr>
<td>27</td>
<td>108</td>
<td>$1^2 + 1^2 + 5^2 + 9^2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$1 + 2\omega_1 + 2\omega_9 - 2\omega_6$</td>
<td>$1 + 2\omega_7 + 2\omega_{12} - 2\omega_6$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>$1 + 2\omega_1 + 2\omega_3 - 2\omega_4$</td>
<td>$1 + 2\omega_2 + 2\omega_{12} - 2\omega_{10}$</td>
<td>$1 + 2\omega_7 + 2\omega_8 - 2\omega_6$</td>
<td>$1 + 2\omega_5 + 2\omega_{13}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>$1 + 2\omega_9 + 2\omega_{13} - 2\omega_5$</td>
<td>$1 + 2\omega_3 - 2\omega_{10}$</td>
<td>$1 + 2\omega_2$</td>
<td>$1 + 2\omega_1 + 2\omega_8 + 2\omega_7$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>$1 + 2\omega_2 + 2\omega_5 - 2\omega_7$</td>
<td>$1 + 2\omega_9 - 2\omega_{10} - 2\omega_{11}$</td>
<td>$1 + 2\omega_3 - 2\omega_4 - 2\omega_6$</td>
<td>$+2\omega_{11} - 2\omega_{12}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>$1 + 2\omega_9 - 2\omega_4$</td>
<td>$1 + 2\omega_{11} - 2\omega_5 - 2\omega_7$</td>
<td>$1 + 2\omega_8 + 2\omega_{13} - 2\omega_1$</td>
<td>$1 - 2\omega_2 - 2\omega_{12}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>No solutions.</td>
<td>$1 + 2\omega_1 - 2\omega_4 - 2\omega_6$</td>
<td>$1 + 2\omega_{10} + 2\omega_{13} - 2\omega_{11}$</td>
<td>$1 + 2\omega_7 - 2\omega_3 - 2\omega_8$</td>
</tr>
</tbody>
</table>

Table C.3: Hadamard matrices of orders 100 and 108 from Williamson Matrices
<table>
<thead>
<tr>
<th>$t$</th>
<th>$n$</th>
<th>$t^2 + n^2 + t^2 + t^2$</th>
<th>$N$?</th>
<th>$\mu_1$</th>
<th>$\mu_2$</th>
<th>$\mu_3$</th>
<th>$\mu_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>116</td>
<td>$1^2 + 3^2 + 5^2 + 9^2$</td>
<td>Yes</td>
<td>$1 + 2\omega_2 + 2\omega_6 + 2\omega_{12}$ $-2\omega_4 - 2\omega_8 - 2\omega_{11}$ $1 + 2\omega_7 + 2\omega_{10} - 2\omega_3$ $-2\omega_5 - 2\omega_8$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>124</td>
<td>$1^2 + 1^2 + 1^2 + 1^2 + 1^2$</td>
<td>Yes</td>
<td>$1 + 2\omega_4 - 2\omega_{10} - 2\omega_{15}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>124</td>
<td>$1^2 + 5^2 + 7^2 + 7^2$</td>
<td>Yes</td>
<td>$1 + 2\omega_4 - 2\omega_{10} - 2\omega_{15}$</td>
<td>$1 + 2\omega_{13} - 2\omega_2 - 2\omega_{14}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>124</td>
<td>$3^2 + 3^2 + 5^2 + 9^2$</td>
<td>Yes</td>
<td>$1 + 2\omega_1 + 2\omega_{14} - 2\omega_{13}$ $-2\omega_6$ $-2\omega_8$ $-2\omega_9$ $1 + 2\omega_{12} - 2\omega_3 - 2\omega_7$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>132</td>
<td>$1^2 + 1^2 + 3^2 + 11^2$</td>
<td>Yes</td>
<td>$1 + 2\omega_2 + 2\omega_5 + 2\omega_{11}$ $-2\omega_6 - 2\omega_8 - 2\omega_9$ $1 + 2\omega_1 + 2\omega_3 + 2\omega_7$ $-2\omega_5 - 2\omega_6$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>132</td>
<td>$1^2 + 1^2 + 7^2 + 9^2$</td>
<td>Yes</td>
<td>$1 + 2\omega_3 + 2\omega_{14} - 2\omega_2$ $-2\omega_{10}$</td>
<td>$1 + 2\omega_1 + 2\omega_3 + 2\omega_7$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>132</td>
<td>$1^2 + 5^2 + 5^2 + 9^2$</td>
<td>Yes</td>
<td>$1 + 2\omega_1 + 2\omega_{10} - 2\omega_8$ $-2\omega_{15}$</td>
<td>$1 + 2\omega_4 + 2\omega_{13} - 2\omega_7$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>132</td>
<td>$3^2 + 5^2 + 7^2 + 7^2$</td>
<td>Yes</td>
<td>$1 + 2\omega_5 + 2\omega_{12} - 2\omega_7$ $-2\omega_{15}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.4: Hadamard matrices of orders 116, 124 and 132 from Williamson Matrices
<table>
<thead>
<tr>
<th>$t$</th>
<th>$n$</th>
<th>$\mu_1^2 + \mu_2^2 + \mu_3^2 + \mu_4^2$</th>
<th>$N$?</th>
<th>$\mu_1$</th>
<th>$\mu_2$</th>
<th>$\mu_3$</th>
<th>$\mu_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>140</td>
<td>$1^2 + 3^2 + 3^2 + 11^2$</td>
<td></td>
<td>No solutions.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>140</td>
<td>$1^2 + 3^2 + 7^2 + 9^2$</td>
<td></td>
<td>No solutions.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>140</td>
<td>$3^2 + 5^2 + 5^2 + 9^2$</td>
<td></td>
<td>No solutions.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>148</td>
<td>$1^2 + 1^2 + 5^2 + 11^2$</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1 + 2$\omega_1$ + 2$\omega_3$ + 2$\omega_5$ + 2$\omega_{10}$ + 2$\omega_{17}$ + 2$\omega_{18}$ - 2$\omega_4$ - 2$\omega_9$ - 2$\omega_{12}$ - 2$\omega_{15}$ - 2$\omega_{16}$</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>148</td>
<td>$1^2 + 7^2 + 7^2 + 7^2$</td>
<td></td>
<td>1</td>
<td>1 + 2$\omega_5$ + 2$\omega_7$ - 2$\omega_1$ - 2$\omega_2$ - 2$\omega_6$ - 2$\omega_{12}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>148</td>
<td>$3^2 + 3^2 + 3^2 + 11^2$</td>
<td></td>
<td>No solutions.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>148</td>
<td>$3^2 + 3^2 + 7^2 + 9^2$</td>
<td></td>
<td>No solutions.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>148</td>
<td>$5^2 + 5^2 + 7^2 + 7^2$</td>
<td></td>
<td>$1 + 2\omega_2 + 2\omega_{15} + 2\omega_{17}$ - 2$\omega_{13}$ - 2$\omega_{14}$</td>
<td>$1 + 2\omega_9 + 2\omega_{12} + 2\omega_{16}$ - 2$\omega_4$ - 2$\omega_{10}$</td>
<td>$1 - 2\omega_3 - 2\omega_{18}$</td>
<td>$1 + 2\omega_8 + 2\omega_{11} - 2\omega_1 - 2\omega_5 - 2\omega_6 - 2\omega_7$</td>
</tr>
</tbody>
</table>

Table C.5: Hadamard matrices of orders 140 and 148 from Williamson Matrices
<table>
<thead>
<tr>
<th>t</th>
<th>n</th>
<th>$\mu_1^2 + \mu_2^2 + \mu_3^2 + \mu_4^2$</th>
<th>N?</th>
<th>Solution</th>
<th>t</th>
<th>n</th>
<th>$\mu_1^2 + \mu_2^2 + \mu_3^2 + \mu_4^2$</th>
<th>N?</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>12</td>
<td>$1^2 + 1^2 + 1^2 + 3^2$</td>
<td>1-</td>
<td>--</td>
<td>13</td>
<td>52</td>
<td>$3^2 + 3^2 + 3^2 + 5^2$</td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1-</td>
<td>--</td>
<td></td>
<td></td>
<td></td>
<td>1-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1-</td>
<td>--</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1-</td>
<td>--</td>
<td></td>
<td></td>
<td></td>
<td>1-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1-</td>
<td>--</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>$3^2 + 3^2 + 1^2 + 1^2$</td>
<td>1-</td>
<td>----</td>
<td>13</td>
<td>52</td>
<td>$5^2 + 5^2 + 1^2 + 1^2$</td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1-</td>
<td>----</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td>7</td>
<td>28</td>
<td>$3^2 + 3^2 + 3^2 + 1^2$</td>
<td>11-</td>
<td>1-</td>
<td>13</td>
<td>52</td>
<td>$3^2 + 3^2 + 1^2 + 3^2$</td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td>7</td>
<td>28</td>
<td>$5^2 + 1^2 + 1^2 + 1^2$</td>
<td>1-</td>
<td>----</td>
<td>13</td>
<td>52</td>
<td>$5^2 + 5^2 + 1^2 + 1^2$</td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1-</td>
<td>----</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td>9</td>
<td>36</td>
<td>$1^2 + 1^2 + 5^2 + 3^2$</td>
<td>11-</td>
<td>1-</td>
<td>15</td>
<td>60</td>
<td>$5^2 + 5^2 + 1^2 + 3^2$</td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td>9</td>
<td>36</td>
<td>$3^2 + 3^2 + 3^2 + 3^2$</td>
<td>11-</td>
<td>1-</td>
<td>15</td>
<td>60</td>
<td>$3^2 + 7^2 + 1^2 + 1^2$</td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td>11</td>
<td>44</td>
<td>$1^2 + 3^2 + 3^2 + 5^2$</td>
<td>11-</td>
<td>1-</td>
<td>15</td>
<td>60</td>
<td>$3^2 + 7^2 + 1^2 + 1^2$</td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>1-</td>
<td></td>
<td></td>
<td></td>
<td>11-</td>
<td>11-</td>
</tr>
</tbody>
</table>

Table C.6: Hadamard matrices from Williamson matrices for orders 12 through 60; alternative representation
<table>
<thead>
<tr>
<th>( t )</th>
<th>( n )</th>
<th>( \mu_1^2 + \mu_2^2 + \mu_3^2 + \mu_4^2 )</th>
<th>( N? )</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>68</td>
<td>( 5^2 + 5^2 + 3^2 + 3^2 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>76</td>
<td>( 7^2 + 3^2 + 3^2 + 3^2 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.7: Hadamard matrices from Williamson matrices for orders 68 and 76; alternative representation
<table>
<thead>
<tr>
<th>$t$</th>
<th>$n$</th>
<th>$\mu_1^2 + \mu_2^2 + \mu_3^2 + \mu_4^2$</th>
<th>$N?$</th>
<th>Solution</th>
</tr>
</thead>
</table>
| 21 | 84 | $5^2 + 5^2 + 5^2 + 3^2$ | | 111-1--111--111--1-11  
111-1--111--111--1-11  
111-111-1----1-111-11  
1-1-----1-1111-1-----1-- |
| | | | | 11111--11-----11---1111  
111-1-1-1111-1--1-11  
111--1-1111--111-1--11  
1----1-1-11-11-1-1-- |
| 21 | 84 | $3^2 + 1^2 + 7^2 + 5^2$ | | 1-1--111------111--1--  
1-11-11-11-11-11----1-  
1-1------11-11------1--  
1111-1-11-----11-1-111 |
| | | | | 1-1--111------111--1--  
1-1-1-1111-11-1-1111--1--  
1-1------11-11------1--  
1111--11-1--1-11-1111-- |
| 21 | 84 | $9^2 + 1^2 + 1^2 + 1^2$ | | 111-1-----11-----1-11  
11----11-11--11-11-----1  
1-1------11-------1-1-1  
1111-1-1-11--1-1-1111-- |
| | | | | 11-11111-1--1-111111-1  
11--1-1-11--11-1-1-1--1  
1111-1-1-----1-1-1111--  
1-1-1-1111--111-1111-- |
| 23 | 92 | $3^2 + 3^2 + 7^2 + 5^2$ | | 111-111-1------11111-11  
111----1-11-11-11----1-11  
1111-111-----111111-11  
1-111----1111---1-1111-- |
| | | | | No solutions. |
| 23 | 92 | $1^2 + 1^2 + 9^2 + 3^2$ | | No solutions. |

Table C.8: Hadamard matrices from Williamson matrices for orders 84 and 92; alternative representation
<table>
<thead>
<tr>
<th>$t$</th>
<th>$n$</th>
<th>$\mu_1^2 + \mu_2^2 + \mu_3^2 + \mu_4^2$</th>
<th>N?</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>100</td>
<td>$7^2 + 7^2 + 1^2 + 1^2$</td>
<td>1</td>
<td>1-11-11-11-11-1-1-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-11-11-11-11-1-1-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-11-11-11-11-1-1-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-11-11-11-11-1-1-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-11-11-11-11-1-1-1-1-1</td>
</tr>
<tr>
<td>25</td>
<td>100</td>
<td>$1^2 + 5^2 + 5^2 + 7^2$</td>
<td>1</td>
<td>1-1-11-111-1-1-1-1-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1-11-111-1-1-1-1-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1-11-111-1-1-1-1-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1-11-111-1-1-1-1-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1-11-111-1-1-1-1-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1-11-111-1-1-1-1-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1-11-111-1-1-1-1-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1-11-111-1-1-1-1-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1-11-111-1-1-1-1-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1-11-111-1-1-1-1-1-1-1</td>
</tr>
<tr>
<td>27</td>
<td>108</td>
<td>$7^2 + 7^2 + 3^2 + 1^2$</td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td>25</td>
<td>100</td>
<td>$1^2 + 3^2 + 3^2 + 9^2$</td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td>27</td>
<td>108</td>
<td>$9^2 + 5^2 + 1^2 + 1^2$</td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td>27</td>
<td>108</td>
<td>$3^2 + 3^2 + 3^2 + 9^2$</td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1-1111-1-1111-1-1111-1-1-1</td>
</tr>
</tbody>
</table>

Table C.9: Hadamard matrices from Williamson matrices for orders 100 and 108; alternative representation
<table>
<thead>
<tr>
<th>$t$</th>
<th>$n$</th>
<th>$\mu_1^2 + \mu_2^2 + \mu_3^2 + \mu_4^2$</th>
<th>N?</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>116</td>
<td>$5^2 + 9^2 + 1^2 + 3^2$</td>
<td>116</td>
<td>No solutions.</td>
</tr>
<tr>
<td>29</td>
<td>116</td>
<td>$7^2 + 7^2 + 3^2 + 3^2$</td>
<td></td>
<td>11-1-1-1-111-11111111-1-1-1-1</td>
</tr>
<tr>
<td>31</td>
<td>124</td>
<td>$5^2 + 5^2 + 5^2 + 7^2$</td>
<td>124</td>
<td>No solutions.</td>
</tr>
<tr>
<td>31</td>
<td>124</td>
<td>$5^2 + 9^2 + 3^2 + 3^2$</td>
<td></td>
<td>11-11-1-11-1-11-1-11-1-11-1-11</td>
</tr>
<tr>
<td>31</td>
<td>124</td>
<td>$7^2 + 7^2 + 1^2 + 5^2$</td>
<td></td>
<td>11-11-1-11-1-11-1-11-1-11-1-11</td>
</tr>
<tr>
<td>31</td>
<td>124</td>
<td>$1^2 + 1^2 + 1^2 + 11^2$</td>
<td>124</td>
<td>No solutions.</td>
</tr>
<tr>
<td>33</td>
<td>132</td>
<td>$7^2 + 7^2 + 3^2 + 5^2$</td>
<td></td>
<td>11-11-1-11-1-11-1-11-1-11-1-1</td>
</tr>
<tr>
<td>33</td>
<td>132</td>
<td>$1^2 + 1^2 + 9^2 + 7^2$</td>
<td></td>
<td>111-1-1-111-1-11-1-11-1-11-1-1</td>
</tr>
<tr>
<td>33</td>
<td>132</td>
<td>$9^2 + 5^2 + 5^2 + 1^2$</td>
<td>132</td>
<td>No solutions.</td>
</tr>
<tr>
<td>33</td>
<td>132</td>
<td>$3^2 + 11^2 + 1^2 + 1^2$</td>
<td></td>
<td>111-1-1-111-1-11-1-11-1-11-1-1</td>
</tr>
</tbody>
</table>

Table C.10: Hadamard matrices from Williamson matrices for orders 116, 124 and 132; alternative representation
<table>
<thead>
<tr>
<th>t</th>
<th>n</th>
<th>$\mu_1^2 + \mu_2^2 + \mu_3^2 + \mu_4^2$</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>140</td>
<td>$9^2 + 5^2 + 5^2 + 3^2$</td>
<td>No solutions.</td>
</tr>
<tr>
<td>35</td>
<td>140</td>
<td>$1^2 + 3^2 + 7^2 + 9^2$</td>
<td>No solutions.</td>
</tr>
<tr>
<td>35</td>
<td>140</td>
<td>$11^2 + 3^2 + 3^2 + 1^2$</td>
<td>No solutions.</td>
</tr>
<tr>
<td>37</td>
<td>148</td>
<td>$3^2 + 3^2 + 7^2 + 9^2$</td>
<td>$11-1-1-11-----11-----1-1-----1-1-1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$11-1-1-11-----1-1111-----1-1-----1-1-1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$1-----11-11-1111-----111-11-1-11$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$11111-11111-1-1-----1111-111-1111-11$</td>
</tr>
<tr>
<td>37</td>
<td>148</td>
<td>$11^2 + 3^2 + 3^2 + 3^2$</td>
<td>$1--111-1-----1-----11-----1-----1-1111$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$11111-1-----1-----11-----1-----1-1111$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$1--11-11-11-11-----11-11-----11-11-----11$</td>
</tr>
<tr>
<td>37</td>
<td>148</td>
<td>$7^2 + 7^2 + 7^2 + 1^2$</td>
<td>No solutions.</td>
</tr>
<tr>
<td>37</td>
<td>148</td>
<td>$1^2 + 1^2 + 5^2 + 11^2$</td>
<td>No solutions.</td>
</tr>
<tr>
<td>37</td>
<td>148</td>
<td>$7^2 + 7^2 + 5^2 + 5^2$</td>
<td>$1--------11-1111-1-----1-1111-11-----11-11-----11-11-----1-11$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$1-1-1-1-11-----1-1-1-1-11-11-----1-1-1-1-11-11-----11-11-----1-11-11$</td>
</tr>
</tbody>
</table>

Table C.11: Hadamard matrices from Williamson matrices for orders 140 and 148; alternative representation
Bibliography


[99] Two clicks and you're gone with this greedy new virus. *Sydney Morning Herald*; Saturday June 12 1999; page 1.


[144] Sun Microsystems. *Java Remote Method Invocation — Distributed Computing For Java*. Available from


