Generic Virus Detection

Virus Detection - the state of the art solutions

By William Hsu, Millersville, Maryland

Note: Source code files accompanying article are located on MacTech CD-ROM or source code disks.

About the author

William Hsu is a third-year undergraduate majoring in Computer Science at the Johns Hopkins University. The following article was based on a paper for an independent programming workshop under Dr. Steven Salzberg. His Internet address is hsu@cs.jhu.edu. His research interests include recent advances in computer virus theory and treatment, program synthesis, and randomized and approximation algorithms.

Recently, computer viruses have attracted a high volume of public, media, and scientific attention. This is no surprise considering the explosion in the development rate of computer virus code. Combine this with the fact that current methods for detection of viruses have had limited success. A new approach to the detection of such code is needed. We're going to look at two variations on algorithms originally developed for string matching. Both modifications allow an increased tolerance for variant “strains” of known viruses, especially for the so-called evolutionary class which mutate themselves at predetermined intervals.

First, a randomization scheme can be applied to an established fast substring matching procedure (such as the Boyer-Moore algorithm). This randomization allows mutation-resistant searching. Second, an approximate pattern matching algorithm for a maximum number of differences can be used. The algorithm is modified by weighting the edit distance metric to make it robust to character padding and removal. Both functions are combined to create a generalized detector capable of finding viral clones, whether produced by human authors or by automatic variations.

This article was produced as part of an undergraduate computer science workshop at Johns Hopkins University. First and foremost, I would like to thank Dr. Steven Salzberg, my project advisor, for his guidance, insight, and instruction. Many thanks to the faculty and staff of the JHU Computer Science Department, and to John Norstad, Brian Seborg, Ephraim Vishniac, and Jan Christian van Winkel for their help and comments.

Virus Detection: Classification, Methods, and History

There has been a great deal of interest in the detection of active “viral” code on both Macs and mainframes, especially as members of an interconnected network. Definitions of the set of programs called viruses have been put forth in many recent articles, most notably in the work by Cohen.1 In this work, detection of the virus is simply considered a pattern string to be searched for in a larger text (a possibly infected program).

Current viral detection procedures are classified according to a system put forth by the Computer Virus Industry Association.2 The association divides anti-viral products into three categories: Class 1 antiviruses (“Infection Prevention Products”) halt the virus replication and prevent the initial infection from occurring - an example of such a program for the Macintosh operating system is the cdev (Control Panel Device) Vaccine. Class 2 programs (“Infection Detection Products”) detect infection soon after it has occurred and mark specific components of system segments that have become infected - they do so by periodically inspecting executable files, and may or may not attempt repair of an infected file. An example of a Class 2 program on the Macintosh is the INIT package GateKeeper and GateKeeper Aid. Finally, Class 3 antiviruses (“Infection Identification Products”) identify specific viral strains (see below for more information about this topic) on systems that are already infected and may remove the virus, returning the system to its state prior to infection. This category is most common, although its effectiveness is dependent on the frequency of user invocation. McAfee's MS-DOS program SCAN and the Macintosh Disinfectant series are Class 3 products.

At present, three approaches to viral code detection have been prevalent. The first, viral signature matching, requires information about the virus. Specifically, a signature-based detector requires the virus’ code length and the location of its “contagious” segment (which is essential to its replication and transfer among storage media, computer memory, and networks). Code enumeration, a second technique, involves examining known programs periodically to test whether any unknown segments of code have been added to the original file. It is most effective when applied before each execution of the program. Finally, checksum methods compare the current size of a file and the summed value of its bytes to the same attributes of a known uninfected version - an infection will often change these values. None of the above methods require the use of the code of the virus itself in its entirety, and all three require user action upon discovery of a virus. Usually, the computer user is urged to restore an earlier copy of the infected file or files from backup. In specific cases, disinfection is possible.

Most software developers claim that the absence of the actual viral code from a detection program prevents its reuse by other virus authors. These authors could conceivably design a virus based on the stolen code and thwart the original detection program. For this reason, developers choose against using a whole (deactivated) virus in detectors. An opposing viewpoint suggests that the only defense against viral code that will not inevitably fail is one which does not depend on the secrecy of its internal mechanisms. A conflict has arisen between these two opinions as each side handles sensitive information very differently. The former advocates secrecy regarding: breaches of security, weaknesses in system defenses, and vulnerabilities of protective software (which will be discussed later in this article), while the latter believes revealing such details will attract support from developers of anti-viral programs and prevent unexpected attacks.

MacDTS has argued that attempting to support anti-viral measures is a futile struggle. This viewpoint fails to factor in the practical results approximate methods have had with problems which were once considered intractable. On the other hand, algorithmic detection of all viruses, including those for which no specimens exist, has been established as an intractable problem. It has been shown that to determine whether any given program is infected is undecidable.3 However, examination of the sample code of a virus, once it has been discovered, allows a signature detection schema to be implemented. Extending the use of this necessary information to both increase the range of the detectable viral set and decrease the amount of data required to do so is a logical goal. It stilll makes sense to include established algorithms suited for such detection as part of the process. By tracking down new viruses quickly, their damage can be lessened further than by distributing cures long after the spread is under way.

Viral Variant Detection an Algorithmic Approach

Choosing an appropriate class of algorithms depends on the treatment of the subject data; in this case, it is beneficial to consider viral and program code as text. For this problem, it is reasonable to use string matching procedures, which are specifically oriented toward character sequences with a pre-defined alphabet (such as the ASCII alphabet). In this context, viruses are searched for by two methods similar to the "Find" and spelling check commands in most word processors. The unique aspect of visrus detection, however, requires the search to be repeated many times (for hundreds of small pieces of the virus). The other search, similar to the "Suggest Word" command in a spelling checker, looks for approximate matches rather than exact ones. Virus detection gives character deletions more importance over inserted characters.

We have developed algorithms that will detect known viruses and unknown clones and mutants of those viruses. For our purposes a viral clone is defined as a known

virus which has been modified prior to its release without changing its viral properties in any appreciable way. For instance, its replication and infection techniques and "detonation" effect (damage done when its preset trigger goes off) should remain identical. An example of a clone is the Hpat virus, a first-generation modified version of the nVIR A Macintosh virus. A viral mutant is defined as a known virus to which self-modification parameters have been added which cause it to create successive clones of itself at intervals or upon a trigger event - mutation may occur after release and may or may not be limited to a finite number of iterations. No Macintosh viral mutants currently exist. The term strain is often applied interchangeably to groups of clones, mutants, and even unrelated viruses (developed separately) which share any common feature. Due to this ambiguity, the expression is not used except in reference to sets of viruses previously designated as "strains", such as nVIR A and B. The word variant is applied to mean an altered virus which may or may not be known and whose function may or may not have been changed during modification of its code; a variant is not necessarily a clone, but all viral clones and generations of mutants are variants of their ancestor viruses. Note that viruses of a single strain are not necessarily variants of each other under this definition; an example is the WDEF strain (Macintosh), with substrains A and B - they are so designated merely because they share the same code label.

A primary method of creating clones and mutants is character padding, the addition of code sequences or characters which do not affect the operation of the virus. Safeguards against this technique are presented in the algorithm discussions. A more difficult strategy is the removal of segments from a known virus - of course, this cannot be carried out indefinitely or even for a large portion of the virus. Finally, a virus may be designed to relocate itself once appended onto or inserted into a host program; auto-propagation is formally considered a feature of worms, but shifting code segments is another way of avoiding detection.

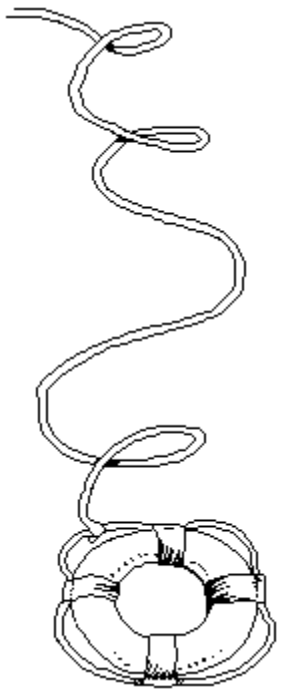
To implement variation-tolerant matching, one of several approaches may be selected. First, approximate string matching for a text of length n, a pattern of length m, and an integer k is among the most common of these. k is the maximum number of differences allowed between a pattern string and the text which is being searched. Algorithms exist which can compute an edit distance, based on the number and type of differences. The "edit" operations are character deletion, insertion, and "twiddle" (transformation of one character to any other). This distance metric can be computed by dynamic programming, a method which breaks down problems which would otherwise require recursion and solves it by computing a table. A straightforward implementation requires O(mn) time; a more complex version solves the problem in O(kn) time4 with some overhead. Parallelization of the procedure allows the values to be computed in O(k) complexity.

The second approach uses a fast substring matching function for small segments of the viral code which is being searched for. The length of each segment is proportional to the expected frequency of variation in the text by addition and deletion of characters. Since the base algorithms and user interface used in this project have been developed elsewhere, our work focuses on general methods for virus detection, rather than implementation issues.

Experimental Input

This code was developed on a UNIX machine before porting to the Mac. Input data at all stages of program development consisted of ASCII and binary data treated as ASCII text (the smallest alphabetic unit was one byte). The "text" used in mainframe testing comprises alphanumeric text, compiled binaries (executable and object files), and ASCII script files. An application was developed for use on the Mac, for which known viruses and their variants exist.5 All text used in the microcomputer development stage consisted of resource data because two primary requirements of viral code - the abilities to replicate and gain control of the operating system - require the execution of the infectious resources.

During the experimental stage, our pattern and text strings were obtained by extracting CODE resource from files found on the average Macintosh hard disk; the segments that were used are listed below. All input data was processed with ASCII character-handling functions. Simple character arrays were used to store both strings. Space requirements were relatively small for the string matching algorithms used. Internal data structures included: a matrix of O(2m) size in the first algorithm - the array requires O(mn) space, but the dynamic programming method only needs two columns at a time - and two arrays for internal computation by the Boyer-Moore algorithm.



The Boyer-Moore Algorithm

The string-matching algorithm developed by Boyer and Moore6 for substring matching has proven significantly faster, in practice, than both straightforward scanning and the finite-state automaton technique implemented by Knuth, Morris, and Pratt. This advantage applies even to binary strings, and becomes increasingly evident as the size of the alphabet increases. Thus, the number of character comparisons per text character scanned is even lower for executables than for alphanumeric text. The Boyer-Moore algorithm employs right-to-left scanning of the pattern string while attempting to find a match within the text body. The main savings are achieved by computing two failure functions which store, for each character in the pattern and the alphabet, respectively, the number of positions to be skipped when a mismatch occurs. Boyer and Moore suggest that entries from both arrays be compared and the larger skip selected. The Boyer-Moore string search requires m+n comparisons in the worst case, and can reliably use n/m steps for large alphabets and short pattern strings.

Our modification of the Boyer-Moore algorithm involved the introduction of a randomized system of string selection. An integer l was chosen to be sufficiently large that an accidental match of a substring of length l was extremely unlikely. We determined this likelihood experimentally (see the discussion below). The pattern source was an original (unmodified) sample of a known virus. Strings of length l were chosen randomly by generating an index between 0 and m-l and designating the next l characters (including the indexed one) of the source string as the pattern P. It was

postulated that this probabilistic factor would establish tolerance for simple changes made to viral code by a potential author in possession of existing code. These changes include:

• Disassembling the viral code and changing variable identifiers.

• Padding null characters or sequences to calibrate the virus checksum.

• Removing small, superfluous amounts of code from the original virus.

• Automatic padding within a viral mutant.

• Pasting viral code segments under new labels or merging segments.

• Reversal of code order using logical jumps.

The probability of a match is experimentally shown (through the tests described below) to be extremely small when a virus is not present - it is clearly possible to discriminate with high precision between infected and uninfected files. An exact match is a very difficult event to duplicate coincidentally; the likelihood of such a match between random strings is infinitesimal even in practice. False positives are relatively rare, though more common than false negatives. Thus, use of a randomized algorithm appears to be a feasible approach to generalized ("inter-clone") viral detection.

Manual "mutation" of code is already becoming commonplace, as is evidenced by the multiple clones of the nVIR strain which already exist on the Apple Macintosh. Simple self-modification has been accomplished in the Core Wars class of programs, and it is not at all unfeasible for a simple virus to be programmed to pad itself with null or checksum-neutral character sequences in an effort to evade detection. Such changes would appear trivial under human inspection. The straightforward searching techniques used in current commercial products, by contrast, are unable to handle even trivial changes. Early efforts to deal with the emergence of viral clones involve omission of parts of the viral signature or selectively summing or enumerating only specified portions of the suspected code.7 This approach lacks generality, however; it is not guaranteed to be proof against even a single revision of a known virus, and is certain to fail against an evolutionary version.8

Important advantages of randomization include the fact that the instructions of viruses need not be physically oriented in their order of execution, but may instead be scrambled by jump instructions (see Figure 1). A second consideration is that preselection of a single segment of code (i.e., the "signature"), as the search pattern,

renders the anti-viral system susceptible to circumvention. Once the identity of the target code is discovered, the procedure may be fooled in one of several ways: by specifically changing or deleting the targeted string; by shifting its physical position; or by disguising it using character padding. Note that none of the above techniques requires real knowledge of how the virus works! A slightly more sophisticated author may easily disassemble the executable code and change certain variable identifiers to thoroughly mask the virus. These variations, in addition to changes made to hide the virus without any detector in mind, may be virtually bypassed when the search string is different for each scan run.

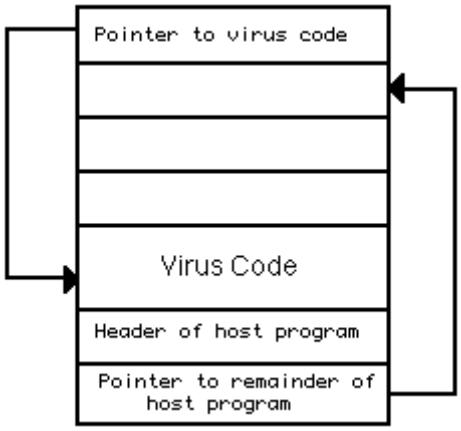


Figure 1

A major concern in refining the probabilistic extension of the Boyer-Moore search is the selection of a string length l. This choice is affected by at least three factors, the most important of which is the chance of a false positive result. Since a false alarm is highly improbable when the pattern and text are unrelated (as is experimentally demonstrated and documented in the tables below), its likelihood is low because the vast majority of legitimate code lacks viral aspects and is dissimilar to the virus search pattern.

Moreover, false alarms are easily avoided by making l as high as possible. On the other hand, l must be made shorter to make the modified Boyer-Moore procedure less sensitive to padding. Figure 2 illustrates a padded clone below an original virus. Each padding sequence insures that at most l - 1 out of m - l strings will fail to be matched, but paddings within l -1 bytes of each other will overlap and “mask” fewer strings. An important feature of drawing random strings from the original virus is that the length of padding sequences is irrelevant; only their frequency must be considered.

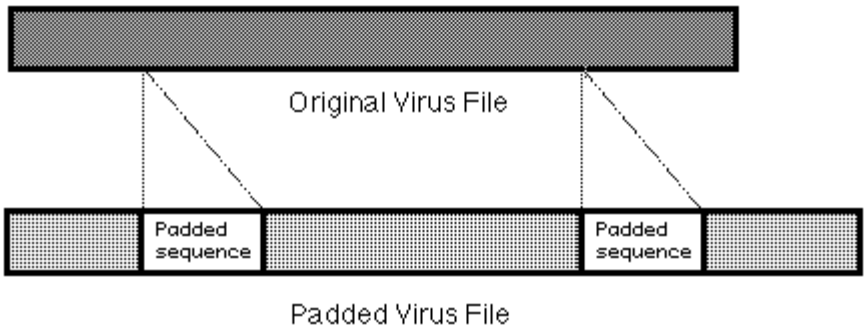


Figure 2

The second factor in determining l is the instruction length of the viral host machine; the unused space in each segment of a binary executable is filled with null (neutral) characters, and a selection of sampled pattern strings containing a high proportion of such characters is likely to contain an excessive number of strings which match with an uninfected text file. One minor weakness will be present regardless of the string length chosen: the virus author will always be able to defeat the randomized filter by increasing padding frequency (although this cannot be done indefinitely). This is one example of the "strength in secrecy" argument in anti-virus programming. On the other hand, the dynamic programming method is reasonably tolerant to padding.

A Dynamic Programming Approach

While randomized string-search algorithms present a viable next step in developing countermeasures to computer virus proliferation, they are only a refinement of the simple straightforward technique. An exact match is required for each string, regardless of how short it may be or how many others are selected and compared. Therefore, it is subject to failure under two conditions, the latter of which results in a false alert.

First, if the length of the randomly selected strings consistently exceeds the distance between padded or removed characters, the algorithm will fail to achieve any matches. Second, the program will erroneously report viruses when the text contains code which is sufficiently similar to the sample virus data to effect more matches than the allowable limit. A heuristic is needed which will deterministically verify or refute the presence of the virus and yield consistent results on every run. Since we are specifically dealing with variants of known viruses, an approximate matching procedure is required.

Fast string matching has traditionally been applied to many text search problems. Where a partial match is available, dynamic programming offers an efficient solution.

The algorithm used in our experiments is a straightforward dynamic implementation which relies on a matrix whose components are computed based on previous entries. The scan function is designed to return the boolean true upon encountering any instance of a k-approximate match between pattern P = p1p2Špm and text T = t1t2Štn for a positive integer k. Assume that n is large relative to m. The following rules are used.9

1. Let Dmxn be a matrix of integers for which D[i,j] equals the minimum number of differences between p1Špi and a segment of T ending at tj.

2. A k-approximate match is detected at any j for which D[m,j] ¾ k.

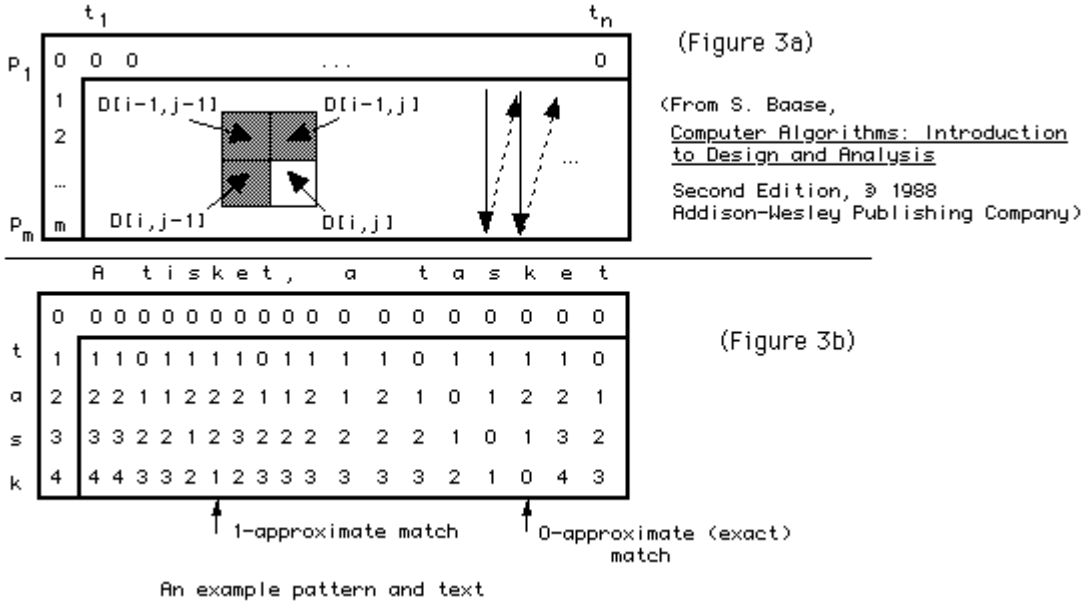
3. The rules for computing D[i,j] consider each of the possible differences that may occur at pi and tj, and the instance for which the two characters match. D[i,j] is assigned the minimum of the following three values:

a) If pi = tj then D[i-1,j-1] else D[i-1,j-1]+a.

b) D[i-1,j]+b (the case where pi is missing from T (deletions)).

c) D[i,j-1]+c (the case where tj is missing from P (insertions)).

a, b, and c are the integer values added whenever a mismatch occurs, and are the central parameters in our modification. Each entry is updated by inspection of the entries above it, to its left, and to its upper left.



(Figure 3a)

(From S. Baase, Computer Algorithms: Introduction to Design and Analysis Second Edition, © 1988 Addison-Wesley Publishing Company)

(Figure 3b)

An example pattern and text

Figure 3

The computation may be done in O(2m) space since only the current and previous columns need to be stored. The work requires O(mn) complexity in the straightforward implementation, but can be achieved in O(kn) time using the improved serial technique by Landau and Vishkin.10 The standard application of string matching by dynamic programming uses a constant value for a, b, and c (for instance, 1). Our method boosts tolerance for padded characters by increasing the ratio between the parameters b and c.

Let r be this ratio, a positive integer; if c is assigned a unit value i, then the matching function may be made tolerant to cases for which the characters in the text are missing from the pattern (i.e., the text has been padded) by setting a and b equal to r •c, making the effective "price" for padded characters lower. A final consideration is the selection of the "threshold" k. It may be determined based on the expected frequency of padding, as is the string length in our randomized Boyer-Moore component. Since r has already been defined relative to i and m, it is a fairly simple task to assign k a value. Typically, it should be close to r (actually, slightly lower to ensure against false alerts) and may be computed using the ratio m/n or simply set to a large fraction of r. Our padding-tolerant implementation uses 1 for i and c, 100 for r, a, and b, and 50 for k. A false match is possible whenever the ratio r is greater than i. However, this only holds in the absolutely worst cases in which an extremely small pattern is matched against a text string of very high length. The probability of this event is equal to that of the consecutive occurrence of all m characters in P within k+m positions of each other. Again, this is experimentally shown to be a statistically rare occurrence, which can reliably be ignored as long as the viral segment length m is not much smaller than r. Generally, a value for r that is higher than the threshold k can be expected to yield few false alarms and will rarely miss a variant created by padding. Conversely, tolerance for missing characters may be effected by increasing the ratio between c and b; in both cases, a is assigned the larger of the two values.

Future applications (and viral threats)

The code below introduces two effective methods of computer virus detection, using newly developed modifications of proven algorithmic techniques. Though previously used in many other applications of computation, these systems are applied here for the first time to the problem of viral code identification. Despite the previously established results on the intractability of universal detection put forth by Cohen, a new class of post-infection scanning methods seems entirely feasible. Further investigation into the circumvention of virus concealment techniques produced experimental results which have supported our assumptions concerning the probabilities of detection and false positives, and support the main premise of both of the algorithms used: that a standard string matching program may be adapted for tolerance toward modification of the text to be scanned. Two significant questions remain concerning general virus detection: First, can clone and mutation detection be extended under a strictly algorithmic foundation to include a broader range of detectable code - especially groups of viruses which have not yet been developed? Second, what optimizations may be performed on the programs to increase speed without sacrificing probabilistic safety? One possible solution is offered through the accompanying program.

Work in string matching, like work in virus detection, is by no means complete. Modern algorithms make use of parallel hardware and improved data structures, such as suffix trees (which may be respectively applied to randomized matching and dynamic programming). Mutating viruses are by no means prevalent yet and have (fortunately) not appeared in the Macintosh operating system. All recent research in “compuvirology”, however, suggests that such programs are feasible and may debut soon - if not on the Mac, then possibly on a larger-scale machine. The viral “visibility” threshold (i.e., the typical size of a virus compared to the average executable size and the machine's general capacity) would even be lower. As an illustration, consider that current viruses approach an order of 10 kilobytes in length and would be considered gigantic if they appeared on machines of 20 years ago. As machine size increases, utilities for virus detection may possess the same precision, but this is not sufficient - they must also match the increasingly sophisticated products of virus authors. Using advances in fast string matching and parallel computing, the software industry can stay not one but many steps ahead of viral attackers.

Code resources used in Randomized

Boyer-Moore Experiments

The randomized Boyer-Moore program was tested on many files to illustrate that on an average Macintosh system, the likelihood of false positives is low. To draw this conclusion, resources from several common Macintosh programs were extracted and searched for variants of nVIR.

Below is a listing of the origin of each code segment used for the generation of Table 2, detailing the size of the file and the application name and resource type (required for Macintosh operating system classification) from which it was extracted.

Segment Length

Group	Number	Source File (bytes)
B	1	MS Word, CODE 1 1474
B	2	Disinfectant 2.0, CODE 7 1116
B	3	Red Ryder 9.4, CODE 37 2164
B	4	SuperPaint 2.0, CODE 42 2480
C	1	WordPerfect, CODE 31 5002
C	2	Font/DA Mover, CODE 1 4670
C	3	WordPerfect File, CODE 1 4542
C	4	ZTerm 0.85, CODE 5 4378
D	1	HyperCard, CODE "HyperTools 2" 26078
D	2	Disinfectant 2.0, CODE 5 18720
D	3	THINK C Debugger, CODE 2 21960
D	4	SuperPaint 2.0, CODE 20 19754

References

[Baase 88] Baase, Sara. Computer Algorithms: Introduction to Design and Analysis. Addison-Wesley, Reading, MA, 1988.

[Boyer 77] Boyer, R.S., Moore, J.S. A Fast String Searching Algorithm. "In Communications of the ACM", pages 762-772. October, 1977.

[Cohen 86] Cohen, Fred B. Computer Viruses. Phd thesis, Electrical Engineering Department, University of Southern California, December, 1986.

[Landau 86] Landau, Gad M., Vishkin, Uzi. Introducing Efficient Parallelism Into Approximate String Matching and a New Serial Algorithm. "In Proceedings of the 18th Annual ACM Symposium on Theory of Computing", pages 220-230. 1986.

[McAfee 88] McAfee, John. Implementing Anti-Viral Programs: Special Report for the Computer Virus Industry Association. Public Information Packet. InterPath Corporation, Santa Clara, CA, 1988.

[McAfee 89] McAfee, John. Computer Viruses, Worms, Data Diddlers, Killer Programs, and Other Threats to Your System: What They Are, How They Work, and How to Defend Your PC, Mac, or Mainframe. St. Martin's Press, New York, 1989.

Listing 1: Dynamic.c

```
/* Dynamic.c - Functions for dynamic k-approximate virus infection detection.
 Copyright © 1992 by William H. Hsu.
 Thanks to John Norstad and Ephraim Vishniac for help and comments.
Portions of this code are based on [Morton 90], which appears in the
May 1990 issue of MacTutor.  Reused with permission.  You may copy, alter,
use, and distribute all code listed here if you leave the file unchanged
up to this line.
 Think C version.

Notes:
•  A main advantage of this code, as explained in the "Methods and History"
section, is that its effectiveness is not diminished by its availability.
 No matter how many potential virus authors read it, the algorithm will
remain equally effective against circumvention.
•  To use this code in your programs:
1. It will be necessary to obtain non-functional but significant (larger
than 300 bytes) resource segments from the virus you are trying to detect.
2. Using a resource editor, insert the viral data under an unused type,
such as 'VDAT', used in the code below -- this will render the virus
code inactive and most likely invisible to conventional (Class 2 and
3) detection programs. As an added security measure, you may wish to
include only code segments above 300 bytes (or a similar threshold length)
to ensure that the virus is crippled.
3. Both the C routines and the Boyer-Moore routines require an expanded
512K Mac or later (specifically, System 3.2 or later); they have been
fully tested on the SE, II, and IIcx.
4. This function should be run upon first launching your application,
or, if it is an operating system utility, during a "dormant" or idle
period.
5. The code below assumes that the VDAT resource contains all 5 segments
of nVIR A; change this accordingly by adding additional virus types (under
a name other than the original infected type) */

#include "dec.h"

FILE *my_file;
void dynamic()
{
  char m[MAXSIZE];
  int pattern_length, index;
  MATRIX table;
  register Handle rsrc;
  short resCount;
  ToolBoxInit();
  CurResFile();
  resCount = Count1Resources ('VDAT');
 /* how many of this type are there? */
  open_file (&my_file, WRITE_MODE);
#ifdef _REPORT /* developer debug flag */
  printf("Searching for <<virus name>>:\n\n");
```

```
  fprintf(my_file, "Searching for <<virus name>>:\n\n");
#endif
  while (resCount) /* loop down to 1 */
  {
 if (resCount == 3)
 {
     printf("\nSearching for <<virus name>>:\n\n");
     fprintf(my_file, "\nSearching for <<virus name>>:\n\n");
 }
   rsrc = Get1IndResource ('VDAT', resCount);
 /* get the resource's handle, but don't load                  it */
 index = SizeResource (rsrc);
 HLock (rsrc);
 /* load next virus segment */
 pattern_length = copy_array (*rsrc, m,                  &index);
#ifdef _REPORT
  printf("Next virus segment loaded (length %d).  Resources left to scan:
%d\n",  pattern_length, resCount);
 fprintf(my_file, "Next virus segment loaded (length %d).  Resources
left to scan:    %d\n", pattern_length, resCount);
#endif
 HUnlock (rsrc);
 initialize(&table, pattern_length+1);
 vResCheck('nVRB', m, pattern_length, table, NO_REPORT);
#ifdef _REPORT
 printf("\n");
 fprintf(my_file, "\n");
#endif
 vResCheck('nVRA', m, pattern_length, table, NO_REPORT);
#ifdef _REPORT
 printf("\n\n");
 fprintf(my_file, "\n\n");
#endif
    --resCount;
  }
  fclose(my_file);
}

void initialize(table, length)
MATRIX *table;
int length;
{
  allocate_table(table, length);
  clear_table(*table, length);
}

void allocate_table(table, size)
MATRIX *table;
int size;
{
  int i;
  *table = (MATRIX)calloc(size, (size_t)sizeof(long *));
  for (i = 0; i < size; i++)
    (*table)[i] = (long *)calloc(2,(size_t)sizeof(long));
}

void clear_table(table, length)
```

```
MATRIX table;
int length;
{
  int i;
  for (i = 0; i <= length; i++)
    table[i][0] = (long)UNIT*i;
}

/* vResCheck - Perform dynamic string search on all resources of a specified
type in the current application. */
void vResCheck (type, m, pattern_length, table, report)
 register ResType type; /* INPUT: type of resource to sum */
 char m[MAXSIZE];
 int pattern_length;
 MATRIX table;
 register short report;
/* INPUT: >0 => report errors with debugger */
 {
 register short resCount;
 /* number of resources of this type */
 register Handle rsrc;  /* resource to check */
 register short oldResFile;
 /* for preserving current resource file */
 register Boolean oldResLoad;
 /* for preserving "ResLoad" flag */
 Boolean found;
 char n[MAXSIZE];
 int text_length, local_count = 1;
 int index;

 /* Switch to the application's resource file.  Note that all resource
calls from here on are the "one deep" calls from Inside Mac, Volume
IV. */
 oldResFile = CurResFile();
 /* remember initial resource file */
 oldResLoad = ResLoad;  /* remember "ResLoad" state */
 resCount = Count1Resources (type);
 /* how many of this type are there? */
  if (report)
  {
 fprintf(my_file, "Text string ");
 printf("Text string ");
  }
 while (resCount)/* loop down to 1 */
 { /* get the resource's handle, but don't load it */
 rsrc = Get1IndResource (type, resCount);
 /* see if it's already in memory */
 if (!rsrc) /* not available? */
 {
 if (report > 0) /* debugging flag */
 DebugStr ("\pResource not available!");
 goto EXIT;
 }
 else
 {
 index = SizeResource (rsrc);
 HLock (rsrc);
```

```
 found = FALSE;
 while ((text_length = copy_array(*rsrc, n, &index)) && (!found))
 {
 fprintf (my_file, "%dŠ ", local_count);
 printf ("%dŠ ", local_count);
 local_count++;
 clear_table (table, pattern_length);
 if (pattern_length <= text_length)
 {
 if (compare (m, n, pattern_length,
 text_length, table))
   found = TRUE;
 }
 }
 HUnlock (rsrc);
 }
 --resCount;/* get next index number */
 } /* end of loop through resources */

 EXIT: /* goto here on tampering or error */
 UseResFile (oldResFile); /* restore original resource file */
 SetResLoad (oldResLoad); /* restore original loading state */
 } /* end of vResCheck() */

/* compare: the actual dynamic programming algorithm, modified to a level
of padding tolerance defined by THRESHOLD */
char compare(p, t, pattern_length, text_length, table)
char p[], t[];
int pattern_length, text_length;
MATRIX table;
{
  long value1, value2, value3;
  int i, j, flip, beep;
  flip = TRUE;
  for (j = 1; j <= text_length; j++)
  {
    table[0][flip] = 0;
    for (i = 1; i <= pattern_length; i++)
    {
 if (p[i-1] == t[j-1]) /* initialize */
        value1 = table[i-1][!flip];
      else
        value1 = (table[i-1][!flip])+UNIT;
      value2 = (table[i-1][flip])+UNIT;
 /* UNIT: the orginal algorithm uses this
 weight for all variations in the text */
      value3 = (table[i][!flip])+EPSILON;
 /* EPSILON: small weighted "distance" --
 as opposed to the single unit */
      table[i][flip] = MIN3(value1, value2, value3);
 /* see discussion of dynamic */
    }
    if (table[pattern_length][flip] <= THRESHOLD)
    {
 if (report)
 {
```

```
      printf("%ld-approximate match found.\n",
table[pattern_length][flip]);
      fprintf(my_file, "%ld-approximate match found.\n",
table[pattern_length][flip]);
 }
 return (TRUE);
    }
    flip = !(flip);/* only an O(2m)-sized array is needed to simulate
a "matrix", because only 2 columns are used */
  }
  return (FALSE);
}

/* data.c: data structure operations (static allocation) for dynamic
AND Boyer-Moore algs */

int read_array(fp, array)
FILE *fp;
char array[];
{
  char c;
  int n;
  n = 0;
  while(((c = fgetc(fp)) != EOF) && (n <     MAXSIZE)) /* Read one element
*/
  {
    array[n] = c;
    n++;
  }
  if (c != EOF)
    ungetc(c, fp);
  return(n);
}

/* fileops.c : file operations for dynamic */

#include "dec.h"
#include "errors.h"

char open_file(fp, operation)
FILE **fp;
char *operation;
{
  SFReply reply;
  char filename[BUFSIZ];
  GetfileName(&reply);
  strcpy(filename, (char *)reply.fName);
  *fp = fopen(filename, operation);
  if (!(reply.good)) {
    fprintf(stderr, CANNOT_OPEN_FILE, filename);
    exit_cleanly(NO_ERROR, EXIT_FAILURE);
  }
  else return(TRUE);
}

/* The following routines deal with the filea. This is all using the
Macintosh HFS. */
```

```
/* GetfileName: read a file name usign the HFS */
GetfileName(reply)
SFReply *reply;
{
 Point  dlgPoint;
 Str255 defName = "\pDynamic Output";
 int  numTypes = 1;
 dlgPoint.h = 100; /* position of the 'open' dialog box */
 dlgPoint.v = 100;
 SFPutFile (dlgPoint, "\pSave output file asŠ", defName, NIL_POINTER,

   reply);
 PtoCstr ((char *) (*reply).fName);
 /* convert from PASCAL to 'C' string */
}/* GetfileName */

/* dec.h - dynamic definitions and declarations */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#define FALSE    0
#define TRUE1
#define NO_REPORT-1

#define K 1024
#define MAXSIZE  8*K
/* All of the tweaking is done here */
#define UNIT1
#define EPSILON  1
#define THRESHOLDUNIT*50

#define READ_MODE"r"
#define WRITE_MODE "w"
#define APPEND_MODE"a"

#define MIN2(a, b) (((a) < (b)) ? (a) : (b))
#define MIN3(a, b, c)((MIN2((a), (b)) < (c)) ? MIN2((a), (b)) : (c))
#define MAX2(a, b) (((a) > (b)) ? (a) : (b))

#define NIL_POINTER0L
#define NIL_STRING "\p"
#define IGNORED_STRING  NIL_STRING
#define NIL_FILE_FILTER   NIL_POINTER
#define NIL_DIALOG_HOOK NIL_POINTER
#define VDAT_RES_ID0

typedef long **MATRIX;

void initialize(), clear_table(), vResCheck(), allocate_table(),
error_message(),
exit_cleanly(), main();
char open_file(), compare();
int read_array();
```

```
int copy_array(array1, array2, bytes_left)
char array1[], array2[];
int *bytes_left;
{
  int bytes_gotten = 0;
  if (!(*bytes_left))
    return (FALSE);
  if (*bytes_left < MAXSIZE)
  {
    memmove ((void *)array2, (void *)array1,       (size_t)(*bytes_left));
    bytes_gotten = *bytes_left;
    *bytes_left = 0;
  }
  else
  {
    memmove ((void *)array2, (void *)array1, (size_t)MAXSIZE);
    bytes_gotten = MAXSIZE;
    *bytes_left -= MAXSIZE;
  }
  return (bytes_gotten);
}
```

Listing 2: BoyerMoore.c

```
/* BoyerMoore.c - Functions for fast, variable- randomized virus infection
detection.
 Copyright © 1992 by William H. Hsu.
 Think C version.

 Notes:
 • As explained in the dynamic algorithm code, these routines are tolerant
toward a wide variety of variations, including padded and mutating viral
code

/* byrmoore.c: main searching file */

#include "dec.h"
void boyer_moore()
{
  FILE *my_file;
  char n[MAXSIZE], *sub_string, **pattern_array;
  int text_length, i, j, sum, match_count, size, divisor, total_match,
index, index2, vdat_count,  refNum, files_to_scan = 5, total_virus_length;
  int *pattern_length_array, *pattern_index_array; /* virus segment lengths
and delimiters */
  long file_size, total_file_size;
  register Handle rsrc, rsrc2;
/* Note: the code which scans files in the same way that Disinfectant
does is far too long to include in this article.  The array used below
is for the purpose of example only.  John Norstad has made the enumeration
part of his program publicly available (by FTP at acns.nwu.edu) */
  Str255 ResFileArray[5] = {"\pOne*", "\pTwo*",          "\pThree", "\pFour",
"\pFive"};
  Str255 DescriptionArray[5] = {"File 1\t", "File 2\t",        "File
3\t", "File 4\t", "File 5\t"};
  ResType typeName;
  short resCount, typeCount, resCount2;
```

```
  srand((unsigned)time(NULL));
  ToolBoxInit();
  open_file (&my_file, WRITE_MODE);
  csettabs (TABS, stdout);
#ifdef _REPORT
  printf("File description\t\t\tScore\tFile size\tAlgorithm's Decision\n");
 printf("================\t\t\t=====\t========\t==================\n\n");
  fprintf(my_file, "File description\t\t\tScore\tFile size\tAlgorithm's
Decision\n");
  fprintf(my_file,
"================\t\t\t=====\t========\t==================\n\n");
#endif
  sub_string = (char *)calloc(SIZE, sizeof(char));
  CurResFile();
  resCount = Count1Resources ('VDAT');
  vdat_count = resCount;
  pattern_length_array = (int *)calloc(resCount, sizeof(int));
  pattern_index_array = (int *)calloc(resCount, sizeof(int));
  pattern_array = (char **)calloc(resCount,
 sizeof(char *));
  while (resCount) /* loop resCount down to 1 */
  {/* get handle, but don't load it */
    rsrc = Get1IndResource ('VDAT', resCount);
    index = SizeResource (rsrc);
    HLock (rsrc);
    pattern_array[resCount-1] = (char *)           calloc(index,
sizeof(char));
    pattern_length_array[resCount-1] = copy_array (*rsrc,
pattern_array[resCount-1],
&index);
    pattern_index_array[resCount-1] = ((resCount < vdat_count) ?
(pattern_length_array[resCount-
 1] + pattern_index_array[resCount]) : pattern_length_array[resCount-1]);
    HUnlock (rsrc);
    --resCount;
  }
  total_virus_length = pattern_index_array[0];

  SetResLoad (true);
  for (i = 0; i < files_to_scan; i++)
  {
    refNum = OpenResFile (ResFileArray[i]);
    match_count = 0;
    divisor = 0;
    for (j = 0; j < ITERATIONS; j++)
    {
      total_file_size = 0;
      sum = 0;
      while (!random_string(pattern_array, sub_string, pattern_index_array,
 total_virus_length, SIZE, vdat_count));
      typeCount = Count1Types ();
      while (typeCount)
      {
        Get1IndType (&typeName, typeCount);
        resCount2 = Count1Resources (typeName);
        while (resCount2)
        {
```

```
          rsrc2 = Get1IndResource (typeName, resCount2);
          index2 = SizeResource (rsrc2);
          file_size = 0;
          HLock (rsrc2);
 while (text_length = copy_array (*rsrc2, &index2))
          {
            compare(sub_string, n, SIZE,text_length, &sum);
            file_size += text_length;
          }
          HUnlock (rsrc2);
          total_file_size += file_size;
          --resCount2;
        }
        --typeCount;
      }
      divisor++;
      if (sum)
        match_count++;
    }
#  ifdef _REPORT
    printf("%s\t\t", DescriptionArray[i]);
    fprintf(my_file, "%s\t\t", DescriptionArray[i]);
    printf("%d\t\t%ld\t", match_count, total_file_size);
    fprintf(my_file, "%d\t\t%ld\t", match_count, total_file_size);
#endif
    if (match_count >= (divisor/LIMIT))
    {
#ifdef _REPORT
      printf("\t%s\n", INFECTED_STRING);
      fprintf(my_file, "\t%s\n", INFECTED_STRING);
#endif
    }
    else
    {
#ifdef _REPORT
      printf("\t%s\n", CLEAN_STRING);
      fprintf(my_file, "\t%s\n", CLEAN_STRING);
#endif
    }
    CloseResFile(refNum);
  }
  printf("\nScore represents matches out of %d, with %d needed to diagnose
infection.\n",   divisor, divisor/LIMIT);
  fprintf(my_file, "\nScore represents matches out of %d, with %d needed
to diagnose infection.\n", divisor, divisor/LIMIT);
  free(sub_string);
  fclose(my_file);
}

char random_string(string_array, sub_string, index_array, length,
substring_length,
vdat_count)
char **string_array, sub_string[];
int index_array[], length, substring_length, vdat_count;
{
  int location, segment, i, zero_count = 0;
  Boolean legal = false, In_The_Right_Segment = false;   /* length and
```

```
segments okay? */
  segment = vdat_count-1;
  while (!legal)
  {
    location = (int)((rand()/(double)MAXINT)* (length - substring_length));
    In_The_Right_Segment = false;
    while (!In_The_Right_Segment)
    {
      if (location <= index_array[segment])
      {
        In_The_Right_Segment = true;
        if (location <= (index_array[segment] - substring_length + 1))
          legal = true;
        else
          legal = false;
      }
      else
        segment--;
    }
  }
  if (segment < vdat_count-1)
    location -= index_array[segment+1];
  for (i = location; i < location + substring_length; i++)
  {
    sub_string[i-location] =  (string_array[segment])[i];
    if (!string_array[segment][i])
      zero_count++;
  }
  if (zero_count < substring_length/2)
    return(TRUE);
  else
    return(FALSE);
}

/* compare: the heart of the Boyer-Moore heurstic, similar to Knuth-Morris-
Pratt's
matching engine */
void compare(p, t, pattern_length, text_length, sum)
char *p, *t;
int pattern_length, text_length, *sum;
{
  ALPHABET_ARRAY char_jump;
  int *match_jumps, print;
  allocate_array(&match_jumps, pattern_length);
  compute_jumps(p, char_jump, pattern_length-1);
  compute_match_jumps(p, &match_jumps, pattern_length);
  if (bm_match(p, t, char_jump, match_jumps, pattern_length, text_length))
 (*sum)++;
  free(match_jumps);
}

void allocate_array(array, size)
INDEX_ARRAY array;
int size;
{
  *array = (int *)calloc(size, sizeof(int));
}
```

```
/* the bad-character failure function
NOTE: if the ASCII alphabet, which has size 256, is
used, this function is not worth computing for resource text strings
of length ¾ 256 */
void compute_jumps(p, char_jump, length)
char *p;
ALPHABET_ARRAY char_jump;
int length;
{
  int c, k;
  for (c = 0; c < CHARS; c++)
    char_jump[c] = length;
  for (k = 0; k < length; k++)
    char_jump[POSITIVE(p[k])] = length-k-1;
}

/* implementation of pseudocode from [Baase 88]
   - uses the good-suffix failure function */
void compute_match_jumps(p, match_jump, pattern_length)
char *p;
INDEX_ARRAY match_jump;
int pattern_length;
{
  int m, k, q, qq;
  int *back;
  allocate_array(&back, pattern_length+1);
  m = pattern_length;
  for (k = 0; k < m; k++)
    (*match_jump)[k] = 2*m-k-1;
  q = m;
  for (k = m-1; k >= 0; k--)
  {
    back[k] = q;
    while ((q < m) && (p[k] != p[q]))
    {
      (*match_jump)[q] = MIN2((*match_jump)[q], m-k-1);
      q = back[q];
    }
    q--;
  }
  for (k = 0; k < q; k++)
    (*match_jump)[k] = MIN2((*match_jump)[k], m+q-k-1);
  qq = back[q];
  while (q < m)
  {
    while (q < qq)
    {
      (*match_jump)[q] = MIN2((*match_jump)[q], qq-q+m-1);
      q++;
    }
    qq = back[qq];
  }
  free(back);
}

int bm_match(p, t, char_jump, match_jump, pattern_length, text_length)
```

```
char *p, *t;
ALPHABET_ARRAY char_jump;
int *match_jump, pattern_length, text_length;
{
  int j, k; /* j indexes text characters; k indexes
 the pattern */
  j = pattern_length - 1;
  k = j;
  while (j < text_length)
  {
    if (k == -1)
      return(TRUE);
    if (t[j] == p[k])
    {
      j--;
      k--;
    }
    else
    {
      j += MAX2(char_jump[POSITIVE(t[j])], match_jump[k]);
      k = pattern_length - 1;
    }
  }
  return(FALSE);
}

/* dec.h - definitions and declarations for bm */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <console.h>

#define TABS4

#define K 1024
#define MAXSIZE  8*K
#define MAXINT   32767
#define MINSUB   8
#define MAXSUB   12
#define STEP4
#define ITERATIONS 1000

#define FALSE    0
#define TRUE1

#define READ_MODE"r"
#define WRITE_MODE "w"
#define APPEND_MODE"a"

#define MIN2(a, b) (((a) < (b)) ? (a) : (b))
#define MIN3(a, b, c)((MIN2((a), (b)) < (c)) ? MIN2((a), (b)) : (c))
#define MAX2(a, b) (((a) > (b)) ? (a) : (b))

#define POSITIVE(a)((abs(a) == (a)) ? (a) : abs(a)+127)

#define CHARS    256
```

```
#define NIL_POINTER0L
#define NIL_STRING "\p"
#define IGNORED_STRING  NIL_STRING
#define NIL_FILE_FILTER   NIL_POINTER
#define NIL_DIALOG_HOOK   NIL_POINTER
#define VDAT_RES_ID0

typedef int ALPHABET_ARRAY[CHARS];
typedef int **INDEX_ARRAY;
typedef ResType **ResTypeHandle;

void compare(), allocate_array(), compute_jumps(), compute_match_jumps(),
error_message(), exit_cleanly(), main();
char open_file(), random_string();
int read_array();
```

Footnotes

1. [Cohen 86] is the most complete and formal of these publications. He gives a full definition of the term virus and technical discussion of worm propagation and viral spread.

2. An inter-corporation group comprised of personal computer industry professionals (generally hardware and software developers) which is devoted to the distribution of anti-viral information (e.g., training seminars and publications) and tracking of new viruses. It was founded and is coordinated by John McAfee, the president of InterPath Corporation in Santa Clara, CA. The full text of his classification schema may be found in [McAfee 88].

3. This proof is available in its original form in [Cohen 86]; the doctoral thesis is exclusively published by the micrographics department of the University of Southern California. [Burger 88], [van Winkel 88], and many other works contain versions of this reduction of new virus detection to the halting problem [Turing 36].

4. A brief definition of O-notation, from [Baase 88]:

f(n) = O(g(n)) (f is “order of” g) if and only if there exist c > 0, N > 0, such that f(n) ¾ cg(n) for every n N.

Thus an O(mn)-time implementation requires time proportional to the product of the lengths of the pattern and text strings, in the long run. An O(kn) version requires time proportional to the product of the maximum acceptable number of differences and the length of the pattern.

Our implementation of the dynamic programming algorithm was coded in C, using Pascal-type pseudocode from [Baase 88] (Chapter 6) as a guide. The O(kn) version can be found in [Landau 86], in the 18th annual ACM STOC Proceedings, with more general pseudocode.

5. Among the Macintosh viruses with known variants (both strains and clones) are the following: WDEF, with strains A and B, and nVIR, with very prolific strains A and B, each with multiple clones found under Hpat, MEV#, AIDS, and other resource titles. An explicit definition of the terms “strain”, “clone”, and “viral mutant” as they are used in this article is given in the introduction.

6. The original presentation of the algorithm is given in [Boyer 77], a paper in the October 1977 CACM; again, pseudocode from [Baase 88] (Chapter 5) was used as a guide in our implementation.

7. This is the pivotal concept in [Morton 90], a recent article in MacTutor. The evident weaknesses in this technique are stressed by the author, who recommends user modification of the anti-viral source code as a means of circumventing viral tampering. This comment forebodes the use of expert systems techniques in viral code design; the use of artificial intelligence intermeshed with viral programs has been predicted in [Cohen 86], and is expected to appear as the availability of compiler tools increases and the viral visibility threshold decreases.

8. The evolutionary virus is a largely theoretical program, first proposed in [Cohen 86]; however, mildly evolutionary code (viral and otherwise) already exists in abundance. User modification of an antivirus is nearly certain to leave it “blind” to successive generations of an automatically self-modifying virus.

9. The table computation rules (with the exception of the distance metric modification - a, b, and c replace 1 in each rule) are quoted verbatim from [Baase 88], Section 6.3.

10.The article is [Landau 86], in the Proceedings of the 18th Annual ACM STOC.