A Generic Virus Scanner in C++ *

Technical Report CSD–TR–92–062

Sandeep Kumar Eugene H. Spafford

The *COAST* Project
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907–1398
{kumar,spaf}@cs.purdue.edu

17 September 1992

Abstract

Computer viruses pose an increasing risk to computer data integrity. They cause loss of valuable data and cost an enormous amount in wasted effort in restoration/duplication of lost and damaged data. Each month many new viruses are reported. As the problem of viruses increases, we need tools to detect them and to eradicate them from our systems.

This paper describes a virus detection tool: a generic virus scanner in C++ with no inherent limitations on the file systems, file types, or host architectures that can be scanned. The tool is completely general and is structured in such a way that it can easily be augmented to recognize viruses across different system platforms with varied file types. The implementation defines an abstract C++ class, `VirInfo`, which encapsulates virus features common to all scannable viruses. Subclasses of this abstract class may be used to define viruses that infect different machines and operating systems. The generality of the mechanism allows it to be used for other forms of scanning as well.

1 Introduction

Computer viruses pose an increasing risk to computer data integrity. They cause loss of valuable data and cost an enormous amount in wasted effort to restore

*This paper appears in the Proceedings of the 8th Computer Security Applications Conference, IEEE Press, 1992.

or recreate damaged or destroyed data. Each month new viruses are reported (cf. issues of *The Virus Bulletin* and *Virus News and Reviews*). As the number of viruses increases, we need tools to detect them and eradicate them from our systems. While the problem of detecting, without error, all viruses automatically is intractable[4], it is certainly feasible to detect simple, known viruses.

In this paper we describe a tool — a generic virus scanner in C++ — that can run in a high integrity virtual memory environment and scan for viruses in files. These files can be available either on the tool's host machine itself, available through a locally mounted file system, or visible to it through an appropriate network mount. For example, the testing machine could be a UNIX workstation accessing a DOS filesystem through its floppy disk interface or remote-mounted on the workstation through a network (e.g., Novell or PC-NFS). The rapid proliferation and use of network software in the PC community has already created a need for such interfaces whereby PC mounted file systems and file servers may be accessible to more powerful workstations on the same local area network.

One common definition of a virus is as a segment of machine code that installs a (possibly evolved) copy of itself into one or more larger "host" programs[4].[1] When the program is executed, the code is activated and enables further spread of the virus, or destruction of data, or both. The principal cause of this problem is the almost nonexistent controls in most PC systems that allows user programs to potentially gain complete control of the system. This allows virus code to perform any operation, and to change any code or data.

Looking for viruses is not a simple matter of looking for extraneous code, because it is not always obvious what is extraneous. Recent "stealth viruses" make even this procedure difficult by ensuring that the original contents of an infected file are returned when its contents are requested as data for examination.[2] It is more reliable to test for infected files by using a system that partitions its processes into distinct address spaces by a virtual memory translation. This way we can avoid the effects of stealth and other memory resident viruses on the scanning procedure. Better still would be the use of a scanner on a completely different architecture — one that cannot support the execution or spread of the searched-for viruses. In such an environment, when a virus scanner running as a user program requests bytes from a file for examination, it is assured of the integrity of the bytes from influence by other user programs; in no case can an ordinary user process modify the interrupt vectors of devices or traps leading to system calls.

If one is testing for viruses on a diskette using a machine that runs processes

[1] Other definitions may be found in the collections [7] and [10].

[2] See [8] for a good description of how stealth viruses operate.

in their own virtual memory space, one can be reasonably sure about the integrity of the memory. Similarly, boot diskettes can be just as easily tested because we do not attempt to boot from the diskettes and, in the process, compromise the integrity of the testing machine in any way. In short, by testing on a machine with processes running in their own virtual address space we ensure, with very high confidence, the integrity of the machine on which we are doing the testing. This is similar to doing a high integrity boot of a PC before scanning for viruses on it. Furthermore, if we are able to run our detector in a completely different environment from the one containing the potential viruses, those viruses cannot infect or interfere with our detector.

2 How viruses are detected

In this section we review current methods of virus detection and end with the conclusion that detection by signature scanning still remains the most simple, economical and commonly used tool for virus detection.

2.1 Virus monitors/detection by behavioral abnormality

In this approach to virus detection, the machine is booted from uninfected files and a virus monitor is installed that monitors various activities of the machine while in day-to-day use. The program monitors known methods of virus activity including attempts to infect and evade detection. This may also include attempts to write to boot sectors, modify interrupt vectors, write to system files, etc.

Software monitors work best when the normal or day-to-day usage characteristics of the system are vastly different from the activity profile of an infected system. This desirable characteristic, however, is not always present. If the virus is cleverly written to always stay within this normal profile, it may be difficult to detect its presence using a monitor. For monitoring to be more effective, users need to be better educated about the behavior and functioning of viruses. They must know how their system works so they can recognize suspicious activity when the software monitor fails.[3]

The chief advantage of a properly implemented monitoring technique is that it works for all viruses — the ones currently known, and the ones yet to be discovered. Furthermore, it can detect infections *before* they occur. Unfortunately, to always detect these infections, the sensitivity of the monitor must be set so high that it

[3][2] and [16] (for example) list practical steps that can be taken to educate users against viral infections.

may generate many false alarms from normal activity. Furthermore, such detectors must be installed at a low-level on the target machine, and must always be run; an infected detector will not be of practical use in preventing further infections!

2.2 Detection by emulation

In this scheme the program under test is emulated by the virus detection program, which attempts to determine the run-time behavior of the program. This is different from monitoring in that the program is not observed while it is actually executing but is emulated with sample input(s). As in the case of monitors, if the program attempts to change the interrupt vectors or open sensitive files it should be deemed suspicious.

Detection by emulation cannot be "precise." That is, we cannot always correctly decide whether the program behaves like a virus. One difficulty in emulating a program is in determining suitable input(s) for it and then emulating it with all the inputs to see if any cause the program to exhibit a virus-like behavior. Timed or context-sensitive behavior may be present that fools the emulator. Another difficulty is in deciding the granularity at which to emulate the program. What is the highest granularity at which emulation can be done to preserve the viral property of a program?

Few emulators for virus detection are in use today. The *VProt* program by Fridrick Skulason for MS-DOS systems has this function as an option.

2.3 Detection by static analysis/policy adherence

This method examines a program to decide whether it meets a prespecified policy requirement, one which may include integrity requirements for the detection of viruses. To determine whether an arbitrary program contains a virus is undecidable [4], but a conservative decision on the presence of viruses may be possible. For example, Maria King [14] describes a technique by which programs can be analyzed to determine whether they meet a prespecified policy.

A policy is specified as a regular expression and defines the characteristics within which the programs being tested must lie in order to fit the policy. The policy may be decided for the entire environment as a whole, may vary from machine to machine, or may be written for clusters of machines. Once a policy is written for an environment, programs running in that environment can be checked to see if their behavior fits the policy. This is a static process, not done by monitoring the executing program, but writing a *minispec* for the program and verifying that the minispec fits the policy. A minispec for a program is also written as a regular expression thus making the verification straighforward. There exist well-known and efficient algorithms to determine whether a regular language is a subset of another regular language. The problem of verification is reduced to finding whether the regular language generated by the minispec is a subset of the regular language generated by the policy. A minispec of a program is a subset of the behavior of the program. This subset comprises the behavior of interest, such as the file manipulation properties of the program. The minispec is written from the source code, design documentation, programmers' notes and the test results of a program.

The chief disadvantage of this approach is that the source of the program is required. The source, if shown to meet the policy requirements, must then be compiled by a *trusted* compiler before being used or else the executable may not accurately reflect the source. Furthermore, this method involves considerable overhead to test and implement. It also requires that all monitored software have a definable minispec. We are unaware of any existing system that uses this mechanism.

2.4 Detection by checksumming

To protect programs against unwanted modification by viruses, a checksum based on the contents of the program is computed and stored in encrypted form within or outside the program. The encryption is done using a one-way function so forgery of a correct checksum after infection is computationally very hard. Before each program is executed its checksum is recomputed, encrypted and matched with the stored result. If the two values are identical the chances of infection are very low; any mismatch implies an unwanted modification.

One problem with this approach is that it requires system support so the check and execution can be performed atomically. [15] presents an excellent survey of checksumming techniques for virus detection. However, the chief argument against checksumming is that it cannot detect viral infection in an already infected file. It can detect further infection but not any current infections prior to generating the first checksum. There is also the danger of infection from what Radai [15] calls an "ambiguity" virus, i.e. infections occurring at the time of copying files or compilations. If an infection occurred just after closing the file being written, the new checksum will be different from the previous one, but there is no way to tell whether infection has occurred.

Checksumming is also susceptible to the "backtrack" attack described in [6]. In a backtrack attack on an executable protected by an encrypted checksum, the executable is disassembled, the virus incorporated into the source and then reassembled to produce a new valid cryptographic checksum. In general the infection can take place at any point prior to the generation of the checksum and one can get to this point from either direction i.e. moving from the source or the object module.

2.5 Dynamic runtime program integrity check

The motivation behind this approach is to ensure the integrity of an executable program while it is running or to detect infections between a program's integrity check and execution. The basic idea behind this approach is to precompute an encrypted checksum for a predefined "granule" of the program. For example, precompute an encrypted checksum for each basic block in the program and store it with the basic block. Every time control flows to the top of the basic block, recompute and match the encrypted checksum with the stored checksum to verify integrity of the instructions in the basic block. Refer to [6] for more information on this approach.

This scheme requires hardware support to run efficiently. Furthermore, unless trusted read and checksum operations are available, this method will not work against stealth viruses. It also does not work well in environments where program files modify themselves to save configuration information, as is the case in most PC operating systems. Again, we know of no system using this method for protection.

2.6 Detection by time stamp modification

This scheme, outlined in [17], is very similar to detection by checksumming in that the time of last modification of the program serves the purpose of a checksum. The time of last modification of a program is usually kept external to the environment storing the program. At regular intervals the timestamps are matched to ascertain integrity of the program. There should not be any means for the virus to get to this file and modify it to destroy evidence of its activity. Another requirement of this scheme is that the time stamping operation must be irreversible by any process in the system. For example, the modification time of a UNIX inode cannot serve as a timestamp mechanism of this type, because the system clock may be set backwards, and because inodes may be written by access to the raw disk.[9]

2.7 Detection by signature scanning

Using the "signature" of a virus to detect its presence in an executable is the simplest and most common approach to known virus detection. Once a virus is isolated, a sequence of bytes (unique sequence) from its code is taken as the identifying string for that virus. Programs (files) are checked for the presence of this signature and flagged as being infected if they contain the sequence.

Signature extraction, however, is a difficult and time-consuming process because it involves disassembling and debugging the infection to identify key portions of the virus. These portions are then combined to form the signature. It is then necessary to test the signature against a large library of programs to reduce the likelihood of false positives occuring when the signature coincidentally matches some production code. Signature scanning is based on the assumption that the virus does not alter itself arbitrarily before infecting another executable. For most viruses prevalent today, regular expressions are sufficiently powerful to capture any such changes.

Simple signatures are usually specified as a string of characters in ASCII: `A923BF7`, for example. Signatures of this form that contain fixed patterns of hex nibbles are simple to use with efficient string matching algorithms. Unfortunately, for some viruses, these fixed patterns are not sufficiently powerful to define the signature in a compact way. For such viruses one would have to specify a large number of fixed patterns to identify a single virus. For example, consider a virus whose code looks like

```
insn
jump PC+1
arbitrary data       (A)
insn
jump PC+1
arbitrary data       (B)
```

and the virus modifies the data at locations A and B before copying itself into another executable. For such a virus, a fixed sequence of hex pattern digits cannot be specified in the signature if the instructions around the arbitrarily modified data were needed in the signature to uniquely identify the virus. For such viruses we may specify the character `?` to signify any value for that position, i.e. equal to the regular expression [0-9A-F] in the signature.

More complicated patterns can be specified to ignore (up to) a specified number of characters or an arbitrary number of values. In general, the specification may be of the form $\{n, m\}$, which means skip at least n characters (nibbles) and at most m values. A specification $\{n\}$ can signify an arbitrary number of nibbles $\geq n$.

The chief disadvantage of scanning as a means of virus detection is its inability to detect unknown or new viruses. Scanning also fails with self-encrypted viruses. It also fails with executables compressed with different compression techniques, making the same virus appear different. Finally, as more viruses are discovered, scanning algorithms tend to run more slowly because they must try to match a larger set of possibilities. Also, the more signatures there are, the more likely that any arbitrary signature will also match some legitimate code in some application.

A further benefit to scanning is that it can also be used against embedded trojan horse code, logic bombs, and other malicious software in addition to simple viruses.[4] All that is needed to detect these pieces of code are appropriate signatures generated from a disassembly. These signatures can be added to the search set and used without any further change to the scanning software.

Cohen argues in [5] signature scanning is not worth pursuing against computer viruses. He (correctly) observes that scanning cannot find new viruses before their patterns are known, nor will such methods work against self-encrypting viruses. He attempts to show that an integrity shell (i.e., checksumming) is the most cost-effective approach to virus protection. The argument in [5] is based on some questionable data and assumptions, however, and is not at all convincing. We believe that the cost-benefit ratio for scanners, either by themselves or in addition to other mechanisms, is much higher than he calculates. This is because of scanners' low impact on existing practice and because of their flexibility. We believe that their widespread use and continued effectiveness in the commercial world affirm this view. Almost all currently available commercial anti-virus tools use signature scanning as their primary detection method.

2.8 Summary of detection methods

Among all the methods of virus detection mentioned above, the simplest and most economical for detecting the majority of current viruses is signature scanning. While signature scanning may not be able to detect all possible viruses, it is still simple and cheap enough to be easily available and useful to the public at large, and it has the least impact on existing code and hardware. Moreover, it is simple to add new patterns to an existing scanner whenever new viruses are discovered. If stealth techniques are thwarted, scanning will work against the majority of common viruses prevalent today.[3] It is for these reasons that we decided to implement a generic signature scanner as our first anti-virus tool.

What follows is a brief description of the C++ classes used to implement our scanner, the interface presented to the user, some measurement results relating to the time and memory utilization of running the scanner on a diskette containing a distribution of file sizes and seeded infections.

[4] See [7], [8], or [10] for definitions of these terms.

3 Description of the tool

We present a brief overview of our scanning tool and give a description of the main C++ classes from which the program is structured. C++ , being an object oriented language, permits data encapsulation and abstraction, and we have tried to take full advantage of these features in partitioning our data and functions. As a result, we expect that future additions and modifications to this program will not change the overall organization of data drastically. We also chose to code in C++ because it will allow us to easily retarget the scanner to different filetypes.

Our program has been written using AT&T C++ 2.1 on a Sun SPARC running SunOS 4.1.1. There are no immediate plans to port it to the GNU G++ version of the C++ language. Such a port should not be difficult, however, as no external libraries have been used, and we have used GNU-compatible features of the AT&T translator.

3.1 A brief description of the classes used in the scanner

Perhaps the most significant class in the scanner is the abstract class `VirInfo` that encapsulates information common to all types of viruses:

```
class VirInfo
{
protected:
   typedef VirInfo *VirInfop;
   String _name; //name of the virus
   Table _aliases; //a table of aliases
   String _sig;
      //signature, stored in hex digits
      //like 07AFB235
   int _infects;
     //interpreted in the derived class

public:
   String& name() { return _name; }
   Table& aliases() { return _aliases; }
   String sig() { return _sig; }
   int& infects() { return _infects; }
   void set_name(char *n) { _name = n; }
   void add_alias(char *n)
   {
      String *t = new String(n);
      _aliases.push(t);
```

```
    }
    void set_sig(char *n) {_sig = n; }

    virtual void print() = 0;
    virtual int infects_ftype(String& fname)
 = 0;
    virtual ~VirInfo() { }
    virtual void read(istream&) = 0;
        //read virus info from input stream
};
```

All storage members of `VirInfo` are protected to allow derived classes access to them. Several public functions, for example `name(), aliases(), sig(), infects()...` are defined in the abstract class. These procedures are generic and the same for all derivations of `VirInfo`. All the undefined functions, except for the destructor, are pure virtual. The input extractor function >> is not a member function and is defined to make a call to the pure virtual function `read()` that must be defined appropriately in derived classes of viruses. The function `read()` therefore encapsulate changes in the input format of virus information. An example of the format we are currently using is:

```
Type:  DOS
Name:  432
Signature:  50CB8CC88ED8E80600E8D900E9040106
Infects:  COM
Aliases:
```

The `Type:` field indicates the type of virus record. For the prototype we only have a DOS derived class of `VirInfo`, but it is not difficult to add newer types in the future. Example extensions would be for Amiga, Macintosh, and Atari file types, and for Unix executables (if a non-experimental Unix virus ever appears).

Note that we can define our pattern alphabet and regular expression format over almost any alphabet we care to specify — it does not matter for the design of the program. We have chosen to define ours over hex digits (*nibbles*) because it is convenient, and because we do not need to compensate for cross-architecture byte order difficulties, as would be the case if we defined over 16 or 32 bit quantities. The patterns provided by others that we used in our testing were defined in terms of hex digits, too.

The scanner starts by reading information about viruses from a signature file containing entries of the type shown above. Depending on the value of the type field, objects representing these virus types (subclasses of `VirInfo`) are created to represent each virus record.

Once objects representing all the virus information in the input file are created, a sparse trie is built to represent it. Each node of the trie is an instance of class `node` (not described in this paper). To each node of this tree is attached a list of viruses (a dynamically resized table with elements of type `VirInfo *`) that match the signature obtained by following the path from the root of the tree (global variable `TreeRoot`) to this node. Each node in the tree has a fan out of $\geq$ 16, corresponding to each hex character [0-9A-F] interpreted as a number plus any regular expressions that can match at this node. For each signature (regular expression), we follow this tree edges from the root (`TreeRoot`), making transitions on each "logical" character (`%n`, `**`, `*n` etc. count as one logical character) of the signature. We create new nodes in the tree on encountering leaf nodes, such that the path from the root of the tree to any given node marks the signature of the virus pattern attached at that node. Usually, leaf nodes specify signatures but if a signature is a prefix of another, then the intermediate nodes can also specify signatures. The algorithm, for inserting *a signature* in this globally accessible tree, in pseudo code is:

```
node *t = root of the tree;

for(every character in the virus signature)
{
   if(the character is a hexadecimal digit)
   {
      make a child of t
         corresponding to this fixed nibble;
      t = this child;
   }
   else if(the character is ?)
   {
      create a child of t labeled ?;
      t = this child;
      /* there may be several marked ?
         if two signatures have a common
         prefix including a ?
      */
   }
   else if(the next two characters are
           of the form %d)
   {
      create a child of t labeled %;
      t = this child;
```

```
      /* there may be several
         marked % similarly */
   }
   else if(the next two characters are
           of the form *d)
   {
      create a child of t labeled *;
      t = this child;
      /* there may be several
         marked * similarly */
   }
   else if(the next two characters are
           of the form **)
   {
      create a child of t labeled **;
      t = this child;
   }
}


add this regular expression at node t;
```

For example, for the following virus entries

```
Type:  DOS
Name:  A
Signature:  012
Infects:  COM SYS

Type:  DOS
Name:  B
Signature:  01A
Infects:  COM ARC
```

the tree looks like in figure 1.

There is also an `ifNibbleStream` class that accesses a file in nibbles (characters) rather than bytes. This class can be used to encapsulate the file I/O interface from the program. Our definition is based on the `ifstream` class available with the AT&T C++ distribution on UNIX. This class can save the current nibble position in the file stream and restore it.

The class `Directory` recurses through a given file system (usually, directory tree) and generates the full pathnames of all the files rooted at the tree. Names

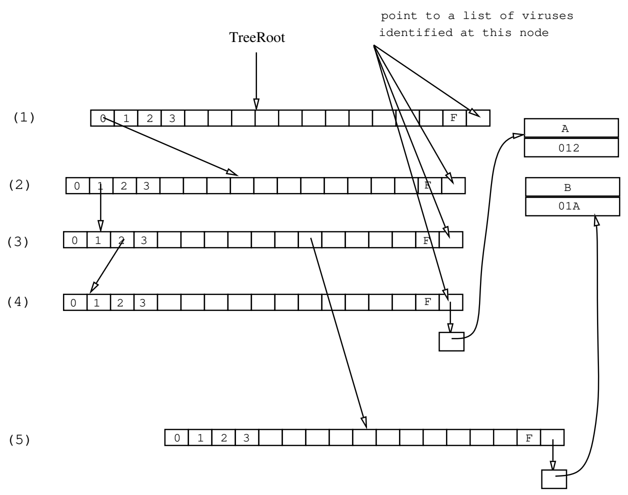

Figure 1: The node tree

of files are returned by successive calls to its member function `next()`. This class encapsulates the file system interface and directory structure visible to the program. These names are then used to open a "NibbleStream" so as to scan the contents of the file.

Thus, whether the file names are `/homes/kumar/foo` or `\homes\kumar\foo` is completely transparent to the program and is completely encapsulated within the classes `Directory` and `ifNibbleStream`. The scanner just requires a sequence of nibbles to do its matching, it is unaware of the file name syntax or the mechanism of opening and closing files. Not only does this allow complete independence from the underlying file system, it allows the scanner to work on encrypted, archived, and compressed files, so long as they can be returned as a stream of 4-bit nibbles and can be recognized during the filesystem scan.

3.2 Why is our scanner *generic*?

Our scanner is *generic* in the sense of and to the extent afforded by an object oriented programming language. We mean that there are no hardcoded dependencies that make it impossible to extend, and that as a principle, the same routines can apply to newer types of viruses, file systems, file formats etc. Whatever modifications may be required to the code are minimal and quite structured.

We believe most of these features are well-supported with our choice of the programming paradigm more than would be afford by simply employing a disciplined programming style in a more conventional language. Consider, for example, how we might add a new virus type *foo* in the scanner. We would derive a subclass *foo* of the abstract class `VirInfo` and define all the pure virtual functions in the abstract class. The routine that reads the file containing the signatures switches on the `Type:` information of the virus to create objects of that type. This routine, of necessity, must be modified to add another type so it can create objects of type *foo* when it encounters a description of a virus of type *foo*. For the most part this is all that needs to be done. All the modification to the code is encapsulated inside the pure virtual functions and defining them for *foo* is all that is required. The pattern matching function will continue to work for *foo*.

Consider now, that the files to be scanned reside in a file system which is quite like the UNIX file system except that the component separator in a file name is `\` instead of `/`. All the changes for this case are encapsulated in the class `Directory` and the rest of the program can probably run unmodified. Where genericity extends encapsulation is in permitting a derived class of `Directory` to be used wherever the class `Directory` was used, thus obviating the direct alteration of the class `Directory`.

3.3 A brief description of the pattern matching algorithm

The following regular expressions are supported in the scanner, the specification is as in TBSCAN. This algorithm is similar to the Aho-Corasick algorithm [1], but it has been extended for wildcard characters.

`?`	match any nibble in the input stream
`%n`	skip 0-n nibbles in the input stream
`*n`	skip exactly n nibbles
`**`	skip an arbitrary number, including 0

The algorithm considers every nibble position in the input stream as a possible beginning of a virus sequence. For each such position, it systematically maintains the set of possible signatures that match as a prefix the input stream nibbles from that fixed position to the current position. Matching stops when the nibble pattern from the fixed position to the current position match any virus signature entirely. Backtracking can occur if signatures contain regular expression patterns `**` & `%n`. Backtracking is currently implemented in a straightforward manner using recursion; failure pointers can be calculated, as explained in [1]. The current algorithm in pseudocode looks as follows:

```
for(each nibble in the input stream)
```

```
   if(traverse(inputNibbleStream, TreeRoot))
   {
      /* virus detected */;
   }

node *traverse(ifNibbleStream& i, node& n)
{
   if(there are virus signatures
      associated with this node)return &n;
   ch = next digit from the input stream;

   pos =  current nibble position of i;

   if(there is a link on ch from n)
      if(ret = traverse(i, the node found by
                   following the link from n
                   on ch))return ret;
   for(all the %d, *d, ? & ** links of node n)
   {
       if(link is of type %d)
       {
            int k = value of d;
            for(int l = 0; l <= k; l++)
            {
               restore file position to pos;
               skip exactly l nibbles;
               if(ret = traverse(i, the node
                            obtained by following
                            the link from n
                            on %d))return ret;
            }
       }
       else if(link is of type *d)
       // all ? are converted to *1
       {
            restore file position to pos;
            skip exactly d nibbles;
            if(ret = traverse(i, the node
                         obtained by following
                         the link from n
                         on *d))return ret;
       }
       else if(link is of type **)
       {
          for(int l = 0; not end of file; l++)
          {
              restore file position to pos;
```

```
            skip exactly l nibbles;
            if(ret = traverse(i, the node
                        obtained by following
                        the link from n
                        on **))return ret;
         }

         restore file position to pos;
      }
   }

   return 0;
}
```

The order in which the regular expression specifications are stored in each node is significant. The search procedure is recursive and always takes the leftmost unvisited path in the search tree when trying to match a regular expression. This ordering ensures that patterns are matched in the order: `[0-9A-F]`, ?, `%n`, `*n` and `**`. While this order does not guarantee the shortest possible match if more than one exists, it does ensure that if the input stream matches a fixed pattern, that pattern will be found first.

For the shortest possible match we would have to place `%n` and `**` before the others because these expressions can match fewer nibbles than `?` or `*n`. The algorithm, being recursive, could then end up trying all possible skips (with **) before matching an exact skip equivalently specified by `*n`, which only appeared later in the node. This is not as efficient, in the general case.

The `node *traverse (ifnibblestream& i, node& n)` routine returns a pointer to the node at which the (partial) signature formed by starting at node `n` and traveling down the tree matches the input nibble stream. If `traverse(ifnibblestream& i, node& n)` returns a non-null pointer, then there is a partial signature beginning at node `n` that matches the input nibble stream. If it returns a null pointer, there is no remaining signature specification that matches the input nibble stream, starting at node `n`.

For example, referring to fig. 1, if the current input nibble stream is `1A9CB` and we call `traverse()` at node 2 then the input stream nibbles `1A` match the partial signature `(0)1A` of virus pattern "A" and a pointer to node 4 is returned. If the inputstream had been `1BA6`... then a pointer to node 5 would be returned, in all other cases NULL would be returned.

When `traverse()` returns successfully to the top level, it specifies a matched signature, not all the matched signatures. We don't look for all possible signatures in the input stream. We just answer the question of infection, i.e., whether the input file contains a virus; we do not enumerate all infections in a file. It is usually adequate to know that a file is infected without knowing the number or types of infections, because one can then discard it and try to restore an uninfected copy of the file, rather than try to disinfect the file of all its infections. If a list of all infections is desired, the scanner may be restarted later in the file, after the current virus identification.

4 A prototype implementation and results

We have constructed a prototype version of this scanner under SunOS to scan for viruses.

Our initial tests have been run on 1197 MS-DOS files known to be infected with viruses. These files occupied 12.3 Mbytes of disk space. We ran the scanner using two signature files. The first was the signature set supplied with TBSCAN program, available from many public ftp sites. The version of this used was the revision of 2/16/92. Our other set of signatures was obtained from the *Virus Bulletin* complete update published in July 1991.

We ran our unoptimized scanner against our set of test infected files. Our preliminary results are summarized in the table below:

No. of sigs	CPU time user + sys	≈ Rate
1	17s	740.9KB/s
(TBSCAN) 347	10m52s	19.3KB/s
(VB) 317	4m35s	45.8KB/s

The CPU time includes the time for any scanner initialization, reading the signature file, generating the node tree and the actual scanning for patterns. The accuracy of virus detection in the files (or "false positives") is not characterized as that is dependent on the quality and number of signatures used; it is not a reflection on the scanning mechanism *per se*.

The initialization overhead of reading in the patterns and storing them in the internal structure is very small — on the order of 7 seconds. As can be seen in the table, running over the entire test tree took only 17 seconds with a single pattern of "?" (match any nibble). Scanning over a larger pattern space takes longer because more patterns must be matched. Furthermore, in the case of more than a single pattern, we needed to read through more of each file before a match was made.

The VB patterns contain fewer wildcard characters, and this likely accounts for the difference in execution times between the two pattern sets. When we implement the optimization of having failure pointers in the scanning routine, we believe the scanning time should decrease signifiantly for both sets of patterns. Furthermore, some work to optimize the file system routines on our system should also result in a significant speedup. We believe that a scanning speed of over 50KB/second against 1000 patterns is possible.

Unfortunately, we do not know how to adequately compare these results with other scanners. The platform and language of implementation for commercial products are undoubtedly generally different and they are not available in source form. Reviews of commercial products published in the *Virus Bulletin* report scanning speeds from under 20 seconds to over an hour for multi-megabyte sets of files on MS-DOS machines (cf., [11, 12] and [13] for recent reviews). However, these measurements seem to be made on fewer total files than our test set, as well as being under different operating systems and file system organizations. For us to fairly make a comparison would require that we port our scanner to a comparable machine and run our tests there, something we may do in the future.

5 Extensions/future work

Our initial implementation was written to test the generality of our design. Although quite promising, we believe that it can be made significantly faster with some further work. As such, we have identified some specific items we would like to address:

- Failure pointers can be computed for each virus pattern so while scanning a file for a particular fixed pattern one does not have to rescan the input to search for virus patterns at every intervening byte in the file. For example if the first n nibbles of a particular virus pattern v (i.e. $v_1 \ldots v_n$) match the file `F` at nibbles $f, f+1, \ldots, f+n-1$ but mismatch at nibble $n+1$ of the pattern, i.e. nibble v_{n+1}, one does not necessarily have to seek for virus matches at nibble $f+1$ of the file because there may not be any viruses with the prefix $v_2 \ldots v_n$. Instead, we only need calculate the appropriate byte beginning the longest matching suffix of the pattern matched so far, and restart the scan with that. With the failure function in place whenever there is a mismatch in the input stream when compared with the longest matching pattern, there will be a change in context and an attempt to restart the scan without rereading the input file. (See [1] for general details.)

- The pattern matcher can be extended on restricted forms of regular expressions specified earlier in the paper. We could use an AO^* search technique to expand the most promising node of the tree to find the virus. This could perhaps lead to a fast probabilistic algorithm for virus detection if we can get good probability values to affix at the tree nodes, perhaps through a large survey of prevalent viral infections.

- New classes `ifArcNibbleStream` & `ifZipNibbleStream` can be derived from `ifNibbleStream` that decode ARC and ZIP files on-the-fly to return decoded nibbles to the scanner. Because of the modular way in which the program is structured, everything else should work without modification.

- Developing more efficient file reading routines, including some that memory-map the input file rather than reading it into intermediary buffers (the method used in the base stream objects).

We also intend to make the code available for beta test so as to gain some further understanding of portability and interface concerns by real users.

6 Conclusions

We believe that a generic scanner program can be an effective and cost-efficient method of virus detection. By constructing a platform-independent scanner, we obtain some automatic protection against stealth and boot viruses that might otherwise make the scanning suspect. Furthermore, by proper definition of the i/o routines, we can scan file system blocks and structures that might not be accessible to a program operating on the system being scanned.

Our approach of using an object-oriented design has proven to be easy to develop and understand. We were able to get the program operational in a short amount of time, and have found it simple to load with several different sets of scan strings. By combining string sets, we expect that coverage may be obtained in a manner superior to most commercial scanners currently available.

We expect that our scanner may prove very useful when released, especially on systems that share multi-platform file systems, and which host archive sites. We expect that by making this a freely-available program, others will contribute modules and scanner strings to increase its usefulness and generality.

Availability

Our eventual goal is to make the scanner program, as well as several associated pattern files for different architectures, available to anyone who wants it. Interested parties are invited to contact the authors for current status and availability information.

Acknowledgements

We would like to extend sincere thanks to Vivek Kalra for helpful comments with an earlier draft of this paper. Vesselin Bontchev of the University of Hamburg Virus Test Center was kind enough to provide us with many megabytes of infected MS-DOS files, and with some DOS-under UNIX tools. Alberto Apostolico aided us in obtaining references to the pattern-matching algorithm. Our thanks to the reviewers for their suggestions and comments.

References

[1] Alfred V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 5, pages 256–300. Elsevier Science Publishers, 1990.

[2] Lisa J. Carnahan and John P. Wack. *Computer Viruses and Related Threats: A Management Guide*. NIST Special Publication 500-166, National Institute of Standards and Technology, 1989.

[3] David Chess. Common viruses. *Virus News and Reviews*, 1:106–107, March 1992.

[4] Fred Cohen. Computer viruses – theory and experiments. *Computers & Security*, 6:22–35, 1987.

[5] Frederick B. Cohen. A cost analysis of typical computer viruses and defenses. In *Safe Computing: Proceedings of the 4th Computer Virus & Security Conference*, pages 737–750. DPMA, 1991.

[6] George I. Davida, Yvo G. Desmedt, and Brian J. Matt. Defending systems against viruses through cryptographic authentication. In *Proceedings of the 1989 IEEE Symposium on Computer Security and Privacy*, pages 312–318, 1989.

[7] Peter J. Denning, editor. *Computers Under Attack: Intruders, Worms, and Viruses*. ACM Books/Addison-Wesley, 1991.

[8] David Ferbrache. *A Pathology of Computer Viruses*. Springer-Verlag, London, 1992.

[9] Simson Garfinkel and Gene Spafford. *Practical Unix Security*. O'Reilly and Associates, 1991.

[10] Lance Hoffman, editor. *Rogue Programs: Viruses, Worms, and Trojan Horses*. Van Nostrand Reinhold, 1990.

[11] Keith Jackson. Product review: Central Point Anti-Virus. *Virus Bulletin*, pages 21–23, May 1992.

[12] Keith Jackson. Product review: SmartScan. *Virus Bulletin*, pages 16–18, July 1992.

[13] Keith Jackson. Product review: Vi-Spy Professional Edition. *Virus Bulletin*, pages 24–26, August 1992.

[14] Maria M. King. Identifying and controlling undesirable program behaviors. In *Proceedings of the 14th National Computer Security Conference*, pages 283–294, 1991.

[15] Yisrael Radai. Checksumming techniques for anti-viral purposes. *Virus Bulletin Conference*, 6:39–68, September 1991.

[16] Eugene H. Spafford, Kathleen A. Heaphy, and David Ferbrache. *Computer Viruses: Dealing with Electronic Vandalism and Programmed Threats*. ADAPSO, Arlington, VA, 1989.

[17] Steve R. White, David M. Chess, and Chengi Jimmy Kuo. Coping with computer viruses and related problems. *International Business Machines Corporation*, 1989.