STATIC ANALYSIS VIRUS DETECTION TOOLS FOR UNIX SYSTEMS [1]

Paul Kerchen, Raymond Lo, John Crossley, Grigory Elkinbard, [2]
Karl Levitt, Ronald Olsson
Division of Computer Science
University of California
Davis, CA 95616

Abstract

This paper proposes two heuristic tools for detecting viruses in a UNIX environment. The tools would be used to detect infected programs prior to their installation. The tools use static analysis and verification techniques. One tool, the *detector*, searches for duplication of operating system calls. A program compiled and linked from source code (such as C) makes calls to standard library routines for operating system services; relevant to detecting viruses are calls on files services, such as open and write. Such object code will contain only one instance of the standard library subroutine for each type of service requested by the program. A virus would most likely carry along its own system calls; hence an infected program would have duplicate calls to the file service and is easily caught by the detector. The second tool, the *filter*, uses static analysis to determine all of the files which a program is capable of writing to. By knowing what files a program can and cannot write, one can decide whether or not that program is suspicious. The paper discusses the features and shortcomings of both tools and gives some implementation details related to the detection of UNIX viruses. In order to defeat these tools, a virus would have to be quite complex and, if successful in avoiding detection by these tools, accept limited propagation. The tools are also useful for detecting more general malicious code, such as Trojan Horses.

1 Introduction

Ideally, one would like to be able to detect an infected program without having to execute it and without noticeably impairing the performance of the system. Some virus detection techniques (see [6] and [7]) rely on run-time checking of program behavior, but employ auxiliary hardware to avoid a performance penalty; the hardware can be viewed as a generalization of the familiar watchdog timer. However, these run-time methods potentially expose the system to a virus which is able to do its damage before being detected. Other run-time techniques (see [2], [3], and [8]) do not allow a program to execute if it fails to pass certain tests; these methods are useful, but they may introduce an unacceptable amount of overhead to the execution time of programs. Typically, these methods involve protecting programs stored on a disk with cryptographic checksums. Another method [10, 11] queries the users at runtime for all file modifications or requires users to identify the programs that can write to his files. Most virus detection techniques have serious limitations because they detect and inhibit the spread of viruses, not their presence. They cannot be applied to programs which are obtained from unreliable sources since they all rely upon having a *clean* copy of the program available for comparison, or they require user interaction at runtime, or they require access protection mechanism absent from most operating systems. Other approaches (e.g.

[1]Supported by grants from Lawrence Livermore National Laboratory, the State of California MICRO program, and Deloitte Touche, Inc.

[2]Currently at Amdahl Corporation, Sunnyvale,CA

virus scanners) cope only with known viruses or virus strains. Our approach attempts to identify viruses through detecting their discerning characteristic in an infected program.

Our approach involves the analysis of a program prior to installation, the analysis attempting to identify suspicious code. By statically analyzing a program, one can in principle determine whether a program contains suspicious code, regardless of whether or not *clean* code is available. This paper presents two static-analysis methods under implementation for detecting suspicious code indicative of a virus. These methods are based on the following premises:

- Source programs are linked with the standard library during compilation. In most systems, the operating system services, e.g. file open, file read, file write, are provided to the user in the form of library functions. Hence a compiled and linked program should contain at most one instance of the trap instruction to the operating system for each system call. Simple viruses, attaching themselves to the beginning or end of a program, would carry along their own trap instructions. Infected programs would have duplication of such trap instructions for some system calls.
- A program containing a virus will contain calls to write the virus to storage, e.g. to the disk, operating system memory, or to uninfected files. Suspicious code, then, could cause the program to write to files the program under investigation is not expected to write to. By enumerating all of the files a program can potentially open, the user of the program is alerted to potentially suspicious code before he runs the program.

These two points form the basis of the two UNIX tools being presented here. The *detector* tool examines a program to determine if it contains any duplicate instances of operating system services (such as file operations like read and write), while the *filter* tool will examine a program to ascertain which files the program can write. These tools are promising because they can detect a large class of viruses and limit the propagation of others. Although these tools are limited by a number of factors, they form a firm foundation upon which more sophisticated tools may be built.

To date, the detector tool has been implemented and tested on several programs with promising results; we have determined that all but one of the UNIX utilities on our Sun-3 workstation running SunOS 3.4 have no duplicate trap instructions. Furthermore, the detector has detected a handcrafted virus that is typical of UNIX viruses. A prototype of the filter is under development, but it has been hand-simulated on several utility programs.

The remainder of this paper discusses the basic approach of the detector and the filter tool. The discussion includes the assumptions attendant to each tool as well as the implications of these assumptions. The implementation of the detector is discussed, giving details about problems and results of experiments performed with it. A discussion of a simple UNIX virus is also given to facilitate the understanding of the implementation. Next, the concepts behind the filter are explained in detail. The shortcomings of each tool are discussed and extensions of the tools are suggested as work for the future.

2 The Detector

2.1 Basic Approach

The purpose of the detector is to identify duplicate calls to operating system services; duplicated calls might be in an executable program and be indicative of a virus that has linked itself to the program. The first step in the detector's analysis is to disassemble a program into its equivalent assembly language representation. The next step consists of finding all instances of code which perform some operating system service. If two different pieces of code are found to contain the

same operating system services, then this condition is flagged as a duplication of services. For most programs, it is reasonable (and necessary) to make the following assumptions:

1. The program uses a standard interface for communicating with the operating system.
2. The program is generated with a compiler.
3. The source program does not call the operating system directly through a trap, instead it uses the operating system interface in the standard library.
4. Virus code can only occur in the code (text) segment of a program.

Assumption one ensures that determination of duplication of services will be relative straightforward. If all programs use the same format for using system services, the detector can always determine what the service is. For instance, in most implementations of UNIX, system calls are performed by pushing the system call number onto the stack and then executing a trap to the operating system. If the system call number is always pushed immediately before the trap is executed, the detector simply has to examine the instruction preceding the trap to determine which service is being used. If a program does not follow such a scheme but instead handles each system call in a different way, the detector must then symbolically execute the program to determine the contents of the stack at the time of the trap instruction—a more difficult and potentially intractable problem. Fortunately, most versions of UNIX use a standard calling scheme. Thus, this assumption is only restrictive for those programs which do not use the standard calling scheme, such as some programs written in assembly language.

The second and third assumption are necessary to ensure that a legitimate, uninfected program will not have any duplication of services. Executable programs linked with the standard library will have one routine which handles all requests for a given operating system service. Any time the program needs a service, it effects the appropriate preparations, such as pushing the other information required for the call (e.g. arguments) onto the stack, and then calls the routine which performs the service. This technique to handle system service call is very common and not confined to UNIX.

For portability and upgrade compatibility reasons, a compiler does not generate code that interface with the operating system directly. Instead, the compiler will treat a system service call as a subroutine provided by the standard library. The actual operating system interface code, i.e. the system trap, resides in the library subroutine. Therefore,the actual interface should appear at most once for each system call in any compiled program. [3]

Finally, assumption four stems from a consideration of file formats and their related restrictions under UNIX. Typically, UNIX uses three file formats for executable files: OMAGIC, ZMAGIC, and NMAGIC. The first, OMAGIC, is obsolete and rarely used. In this format, the text segment is non-sharable and not write protected, so the data segment is immediately contiguous with the text segment. The second, ZMAGIC, is the default format produced by *ld*, the link editor. For this format, the text and data sizes must both be multiples of the page size since the pages of the file are brought into the running image as needed. The third format is similar to the second except the data and text segments are not required to be multiples of the page size; the entire image is preloaded into memory at run time. Most versions of UNIX enforce segmentation of code and data, meaning that executable code and non-executable data must reside strictly in their respective segments. Furthermore, the text segment is not writable during run time and execution of the data segment is not allowed. As a result of these restrictions, a virus which infects a program must do so by placing all of its code into the text segment; it cannot hide any code in other parts of

[3] In order to defeat the detector, a virus would have to use the operating system calls of the program it is attempting to infect, rather than trivially attaching itself to the beginning or end of the program. Later, we discuss ways to catch attempts to defeat the detector.

the file. NMAGIC and OMAGIC format files, therefore, are somewhat more resistant to viruses than ZMAGIC format files since no unused space is available for a virus. However, a virus may still be able to infect such files if it can somehow hide the increase in the size of the host program (perhaps through a flaw in the operating system or by compressing the original code to obtain space). ZMAGIC format files are even more vulnerable. For instance, under SunOS 4.0 the page size is eight kilobytes, meaning the average ZMAGIC format program will have approximately four kilobytes of zero-padded space in both its text and data sections. This space is large enough to hold a fairly complex virus written in assembly language. However, in all three cases, the virus code must still appear in the text segment, making it detectable by the detector. If all of these conditions are met, then the detector can be used to determine if the program under consideration contains any duplication of system calls.

2.2 Implementation and Results

A prototype of the detector has been implemented on a Sun 3 workstation running SunOS 3.4 and has been tested on several of the standard programs from */bin*, */usr/bin*, and */usr/ucb*, but its application is not limited to UNIX systems. This prototype, called Snitch, is written in the C and Icon programming languages and consists of two major modules: the disassembler and the analyzer. The first module, the disassembler, takes an executable program as input and produces the equivalent Motorola 68020 assembly language representation as output. The second module, the analyzer, takes the output from the disassembler and examines it for duplicated code.

For SunOS 3.4, a system call is performed by pushing the system call number onto the stack and then executing a trap instruction. Because the call expects the top of the stack to contain the number of the call to be made, determination of duplication of services becomes straightforward: one only needs to backtrack from the point of the trap to determine the last item pushed on the stack; that item will be the system call number. Furthermore, most of the standard library routines push the system call number immediately before executing the trap, making the analysis phase even simpler. The analyzer reports any duplications found as well as the number of occurrences of all system calls.

The results of the experiments performed on Snitch are as follows. Approximately one hundred programs (mostly UNIX utilities) were tested for duplication of services with some of them infected with a simple virus (described in Section 3.2). All of the infected programs were found to have duplicated system calls, while only one uninfected program was flagged as having duplication of services: */bin/csh* contained two instances each of the *getgid* and *getuid* system calls. One may conjecture that such duplication occurred because of post-linking binary patching. Since the duplicated services were not of a serious nature; for a program as large as the C-shell, such an occurrence should not be surprising or indicative of malicious code.

2.3 A Simple Virus

For purposes of testing Snitch, a simple virus was created which infects SunOS 3.4 executables. The virus is considered simple because it makes no effort to conceal itself and it does not use a sophisticated method for replication and propagation, although it is capable of avoiding multiple infections of the same program. Basically, the virus works as follows: First, the virus determines whether it has previously infected the target program. Under SunOS, executables have a standard header which contains format information, start-up code, a branch to the user's code, and then clean-up code. The format information tells in which format (OMAGIC, ZMAGIC, or NMAGIC) the file is arranged. The start-up code initializes environment variables and other constructs while the clean-up code restores the old environment and makes a smooth return to the shell. All of this information is common to most executables and of a constant length. Therefore, the branch to the main body of code always occurs at a certain offset from the beginning of the text segment.

Furthermore, the user code always immediately follows the clean-up code, making the branch address the same for all programs. Thus, to determine previous infection, the virus simply examines the location in the text segment where the branch instruction occurs (bytes 70-73) and determines if the address is the standard address (20A0 hexadecimal). If it is, the virus commences the infection process.

Next, the virus determines if it has enough space to infect the program without overwriting any legitimate code or increasing the size of the program. The only format which allows any zero-padded space is the ZMAGIC format; if the file is not in ZMAGIC format the virus exits and passes control to the legitimate code. If the file is in ZMAGIC format, the virus determines whether there is zero-padded space at the end of the text segment. This task is accomplished by looking for zero-padded space of length N between the end of program and the end of the text segment, which is multiple of 8K bytes. N is the length of the virus code.

Finally, assuming there is enough room, the virus copies itself from the host program into the target program by copying the last N bytes from the host program's text segment. It then changes the branch instruction in the start-up code so that the virus code is executed after the start-up code and before the legitimate code. Five system calls are used by this virus (open, lseek, read, write, and close) and its length is approximately 150 bytes. A program infected with this virus is easily detected by the detector.

2.4 Limitations of the Detector

The most obvious way of defeating the detector is simply to make the infected program not have any duplication of actual interface to the operating system; if the virus uses the existing services it cannot be detected with the detector. Use of existing services would be simplified if the symbol table information was left in a given program. In this case, a virus could determine the location of the needed services and hook into them, thereby adding only that code which was not already present in the host program. Even without the symbol table, a virus could search the host program, looking for the services it requires. Then, it would import only those services which it could not find.[4] Also the virus could escape detection by inserting a dummy system call that is absent from the uninfected program, pushing the system call number onto the stack and jumping to the trap instruction inserted. Such viruses would escape detection by the current detector, although it could be extended to identify code that searches a program for system calls. We are currently investigating these and other approaches to defeat the detector and to extend the detector to make it more robust.

3 The Filter

3.1 Basic Approach

A *virus filter* is an automatic classifier which applies static analysis techniques to detect the presence of a virus. Since computer viruses multiply by implanting themselves in healthy programs, a necessary condition for propagation is their ability to modify executables. Our approach, although based on the technique of formal verification differs from classical verification. Verification entails proving a program with respect to a specification – a statement of what function the program is intended to compute. For the purpose of detecting suspicious code, we are assuming no specification will be provided. Instead, programmed into the filter is a property to be determined of the program under analysis. For the current version of the filter, the property is "the files that the program could write to". The basic approach is first to identify all open calls in the program and then

[4]This may not be as easy as it sounds, however, since the virus must then know where each of its constituent parts is located within its code as well as how to extract them.

to enumerate the possible filename arguments to these calls. As we demonstrate, the analysis is feasible as only a small fraction of a program is involved in generating filenames. Upon being presented with the names of files that the program could write, the user could determine if the program is suspicious. Of course, a virus could still be present, but its propagation would be severely limited – essentially to just those files. Crocker and Pozzo (see [4]) (hereafter abbreviated to Crocker) proposed a virus filter based on formal specification and verification techniques. But through the following hypotheses, they conjecture that the analysis will be vastly simple than that usually associated with program verification.

Hypothesis 1 It is possible to formulate restrictions for the majority of useful programs such that the restriction is syntactically simple enough to be machine processable and fine-grained enough to represent the full range of authorized modifications made by real programs. A restriction is the specification of the modifications a program makes. It is created by a program developer wishing to submit an executable program for potential use.

Hypothesis 2 It is possible, on the average, to analyze benign programs in a straightforward way.

Hypothesis 3 It is possible to classify modifications such that ordinary changes can be distinguished from suspicious ones.

Generally, we agree with Crocker's hypotheses, but argue that for some programs (benign or infected) the semantic analysis required is more complicated than implied by these hypotheses.

In UNIX systems, the propagation of a virus through direct access to files is through the *open*, *create*, *rename*, *link* and *unlink* system calls. A virus may open and write to an executable or replace an executable by its viral counterpart. Using symbolic evaluation techniques, it is sometimes possible to determine the arguments to these system calls and hence the names of files being modified. The enumeration of the files which may be modified by the program being investigated provides clues to detecting viruses. For example, the program `date` does not write to any files (except standard output). If the enumerated list of files the filter identifies for `date` is not empty, it can be concluded that the `date` program is suspicious. The analysis of the benign `date` program is very straightforward. Much less straightforward is the `split` program. `Split` reads a file and writes it in n-line pieces to a set of output files. The name of the first output file is an argument specified in the command line with "aa" appended, the second one with "ab" appended, and so on. If no output file argument is given, "x" is used as default. The program should only create files starting with the prefix specified in the command line or the default prefix. Therefore, we can say the `split` program is safe if the enumerated files satisfy this restriction.

In general, a program is said to be suspicious when

1. The program's acceptance criteria is not satisfied – there is a high potential for a virus. The acceptance criteria states that the enumerated set of filenames is acceptable to the user.

2. The program is too complex to be evaluated by our filter. No definitive answer is obtained from the filter so the program is not accepted. In practice, it would be the responsibility of the programmer to argue that a suspicious program is not contaminated.

Otherwise the program is said to be safe.

After sampling some commonly-used programs, Crocker concluded that the patterns of filename generation could be classified as follows:

Implied – There is a fixed, possibly empty, list of files to be modified. For example, `date` modifies no file. `vipw` modifies `/etc/passwd`.

Parameters – Filenames are passed to the program as command line arguments. For example, indent indents and formats a C program specified in the command line.

Transformations – Some programs such as compilers and editors create new files based on the arguments in the command line. For example, `compress` transforms `filename` to `filename.Z`.

Temporary files – New filenames are generated independently. For example, `vi` generates temporary files in the `/tmp` directory.

Dialogs – The filename is provided by the user when the program is running. For example, `csh` (a standard UNIX command interpreter) file redirections are obtained from terminal input.

In all of these classifications, the algorithms used to generate filenames are quite simple involve a small fraction of the total program. Since most realistic programs are far too complex to be analyzed in their entirety and most of the code is unrelated to filename processing, our approach is to isolate that part of the program concerned with filename generation and disregard the remaining part. The simplicity of the resulting reduced program should make the static analysis tractable.

In summary, our filter tries to determine the names of all files which might be modified by the program. By comparing the enumeration of names and the specified restriction, the virus filter can claim the program is safe or is suspicious. The complexity of the programs in their entirety may prohibit comprehensive analysis, so part of our method eliminates that part of the program not related to filename processing. We call this method *slicing*. After slicing, the residual program is usually small in size and, thus, analyzable.

A virus in a program could escape detection by the filter if it is content to contaminate only those files for which the program has legitimate access. For example, a virus hiding in the EMACS editor could infect a program being created using the editor. However, once infected this program could infect only those programs its designer has given it access to. Any code in the original virus that would involve writes to other files would be detected by the filter.

3.2 Implementation and Results

This section discusses the implementation of our approach. The input to the virus filter is a binary executable. The output is the enumerated set of the files that may be modified by this executable. The virus filter proceeds through six steps. The first five steps are the preprocessings required to extract the program fragments which contribute to filename generation. The last step involves symbolic execution and analysis. The six steps are as follows:

1. Translation to an intermediate language
2. Determination of basic block and life span
3. Determination of data dependencies
4. Anti-aliasing
5. Slicing
6. Symbolic evaluation and analysis

Given a program to be analyzed, the virus filter first translates it into a C-like intermediate language. Then the filter relabels variables in order to decouple semantically disjoint variables sharing the same storage. Next, the data dependencies are found by analyzing the program syntactically. The filter performs anti-aliasing analysis to unify references to the same storage. Extra

dependencies are added to the data dependency graph when aliased storage is found. Based on the data dependence graph, the program is sliced into pieces. Finally, the pieces which are related to filename processing are extracted and symbolically executed. The filter also applies some theorem proving techniques, primarily to derive inductive assertions for the few, if any, loops involved in filename enumeration.

The following simple example, written in our C-like intermediate language, is used to illustrate the different steps of the virus filter. This example program consists of two independent fragments of code which perform different operations although they share the same variables. It demonstrates our method of decoupling variables by relabeling. Then, we separate it into two independent program fragments by applying slicing. After locating the appropriate fragment containing the system calls, we apply symbolic evaluation and analysis to determine the filenames.

Example: We pick up this example after translation to an intermediate C-like language. x is a filename string, i is an integer, str() is a function converting an integer to a string. Not shown are the *open* system calls, assumed to occur at any line in the program with filename argument x.

Line number	Intermediate code	
1	`i = 1`	
2	`x = ''f''`	
3	`x = x || str(i)`	# string concatenation
4	`i = i + 1`	
5	`if (i <= 3) goto line 3`	
6	`print x`	
7	`i = 200`	
8	`x = str(i)`	

The filenames generated would be:

`f`	if the open system call follows line 2
`f1, f12, f123`	follows line 5
`f123`	follows line 6
`200`	follows line 8.

3.2.1 Translation to Intermediate Language

The input to the virus filter is assumed to be a machine compiled binary program, not an arbitrary assembly language program. In the first stage, the program is decompiled into a machine independent, C-like, intermediate language. We have designed the intermediate language such that analysis attendant to steps 2-6 is simplified. To be specific, the intermediate language contains at most one assignment per statement and control is transferred by the goto statement only. The decompiler recovers semantic information about variables which are lost during the compilation. The goal is to partition memory into regions such that each region is the storage for a simple or structured variable. All storage locations are made explicit and side effects are eliminated. Library calls, like string assignments (string copy) and integer to string conversions, are replaced with defined functions in the intermediate language. Thus the virus filter is more likely to produce intelligible output through reference to higher level functions.

Since our filter is designed to work with binary executables, we need a decompiler to translate machine codes to the intermediate language. Intuitively, the intermediate language should contain more information than the machine code, e.g. concerning types and addresses of symbols. We

should be able to locate extra information concerned with the high-level language from sources such as the symbol table. Even if we cannot find anything directly, we may still be able to deduce data types, procedure entries, etc, from the style in which the compiler generates code.

3.2.2 Basic Block and Life Span Analysis

Variables are often *recycled* in many programs in order to save storage or simply as a matter of programming style. In many programs, variable i is a general purpose loop counter which is reused in different, unrelated parts of the program. This recycling adds dependencies to the dataflow graph that can be eliminated. The elimination involves the renaming of the variables on the left hand side of an assignment statements.

After translating to the intermediate language and relabeling, the program is decomposed into basic blocks for life span analysis. A basic block is a sequence of instructions in which

1. All control transfer statements are at the end of the block.
2. Only the head of a basic block can be the target of any control transfer statements.

The life span of a variable corresponding to an assignment is the span of validity of its value. The life of a variable starts on its assignment and propagates to basic blocks that the current block can lead to. We now pick up the example be derived as the filter starts in step 2. In line 4 of the following table, the value of i at the right hand side may be derived from three possible sources because there are 3 assignments to i (lines 1, 4, and 7). The purpose of life span analysis is to eliminate impossible combinations, i.e. i.7 can never be the i.4 of line 4.

Variables on the left hand side are relabeled uniquely by their name and line number. The program is broken into three basic blocks. The live variables are given in the rightmost columns.

Line number	Intermediate code	Life of i	Life of x
1	`i.1 = 1`		
2	`x.2 = ''f''`	`i.1`	
3	`x.3 = x || str(i)`	`i.1 i.4`	`x.2 x.3`
4	`i.4 = i + 1`	`i.1 i.4`	`x.3`
5	`if (i < 3) goto line 3`	`i.4`	`x.3`
6	`print x`	`i.4`	`x.3`
7	`i.7 = 200`	`i.7`	`x.3`
8	`x.8 = str(i)`	`i.7`	

3.2.3 Finding Data Dependencies

Given the life span of the variables, the syntactic data dependencies can be determined by dataflow analysis. Consider, for example, statement 3 in the example after step 2: "x.3 = x || str(i)". The variables x and i are referenced; x.2 and x.3 are live when x is referenced; i.1 and i.4 are live when i is referenced; x.3 is written to. Thus x.3 depends on i.1, i.4, x.2, and x.3.

Thus for step 3, the dependencies are determined to be:

```
x.3 ← x.2, /* x.3 depends on x.2 */
x.3 ← x.3
x.3 ← i.1
x.3 ← i.4
i.4 ← i.1
i.4 ← i.4
x.8 ← i.7
```

The objective of the data dependency analysis is to slice the program into independent portions to simplify static analysis. Since filenames are usually generated by simple algorithms, syntactic dependencies are considered instead of semantic (real) dependencies. This simplified analysis is adequate for the worst case dependencies. In the above example, we can recursively trace back and find all variables on which x depends at line 8. The dependency subset is found to be "x.8 ← i.7". The subset program is composed of lines 7 and 8 as indicated by line numbers in the dependency subset.

Similarly, the variables x.2, x.3, i.1, i.4 are related to the computation of x at line 6. Lines 1 to 4 constitute the corresponding subset program.

3.2.4 Performing Anti-aliasing

We need to solve the aliasing problem which results from the possibility of referencing a memory location directly through a variable or indirectly through a pointer. Such sharing of storage must be identified before we can have a correct data dependency graph. After the virus filter identifies the aliases, additional dependency arcs are added into the graph. The aliasing is found by considering the pointer assignments. Let us call the variable on the left hand side of the assignment statement the 'home' variable. Reference through a pointer will add a dependency to this variable. Modification through the pointer will add new labels to the home variable. Since the life of the new label must be computed, the virus filter may need to iterate through steps 2 to 4 several times. The iteration stops when no new dependencies are identified.

3.2.5 Slicing

After completing steps 1 to 4, we have the data dependency graph and the next step is slicing to identify the program fragment associated with each open system call. A fragment terminates with an implied system call, the arguments of which are to be determined in step 6.

Continuing with example 1, if the system call immediately follows line 8, the sliced fragment would be:

```
7  i = 200
8  x = str(i)
```

If the system call immediately follows line 3, the slice fragment would be:

```
1  i = 1
2  x = ''f''
3  x = x || str(i)
```

To obtain the pertinent program fragment, the filter traces back from the system call through the data dependency graph to obtain all of the variables the system call depends on, i.e, the line numbers of the relevant program statements. Having the line numbers, we can easily *slice* out the program fragment.

3.2.6 Symbolic Evaluation and Analysis

The sliced program is then symbolically executed to identify filenames generated. Proceeding forward through the statements in a program fragment, each variable obtains a set of values, each value in the set being a value the variable could be assigned in a real execution.

The symbolic evaluation is straightforward when the program has no loops. For a program containing loops, more complicated techniques, as described by German and Wegbreit in [5], are required.

Given the input and output assertions for a loop, four methods to obtain inductive assertions for the program have been proposed: (1) weak interpretation, (2) using loop exit tests and generalization, (3) predicate propagation, and (4) extracting information from unsuccessful proofs. The first three methods can be used in our virus filter. The last one is not applicable because it works backward from the output assertion, which we assume will not be available.

The followings are the salient points of German and Wegbreit's first three methods as they bear on the virus filter:

1. Symbolic evaluation in a weak interpretation.

```
P = start address of S;
{I: start address of S <= P <= end address of S}
while (P < end address of S)
       {I}
       P = P + 1;
       {I}
```

   For example, suppose P is a pointer variable and S is a string variable. P is initialized to the start address of S on entry to a loop; P is incremented on each pass through the loop, and the loop is exited when P is greater than the end address of S. It follows that inside the loop, the inductive assertion I will contain the expression: start address of S $\leq$ P $\leq$ end address of S. Weak interpretation attempts to derive simple facts of this kind; specifically, it considers only simple linear equalities or inequalities relating two variables.

2. Combining assertions with loop exit information.

   Suppose a loop is exited when some test D is true and that after the loop some assertion P is to hold. Since P is to hold after the loop, the assertion D $\rightarrow$ P (read D implies P) must be true inside the loop and just before the exit test. It is very likely that D $\rightarrow$ P is sufficiently strong a loop invariant for our purpose.

3. Propagating valid assertions forward through the program, modifying them as required by the program transformations.

   Whenever an assertion is known to be valid, it is useful to propagate it forward in the program, deriving the strongest consequences of the assertion downstream. Through substitutions, assertions are modified on passing through decisions and assignments to produce their consequences.

   Our preliminary analysis of the filter has determined that these 3 methods are adequate for the analysis of loops involving file enumeration code.

3.3 Example: The Split and the Copy Programs

The program split.c is analyzed. The synopsis of split is

split [-number] [infile] [outfile].

In short, split reads a file and writes it in n-lines pieces onto a set of output files. The name of the first output file is an argument specified in the command line with "aa" appended, the second one is outfile with "ab" appended, and so on; the name generated form a lexicographic sequence. If no argument is given, "x" is used as default. The following is the sliced split program, resulting from applying steps 1-5 of the filter.

```
10 argc = INPUT;  argv = INPUT
16 outfile = "x"
21 for (i = 1; i < argc; i++)
38  outfile = argv[i]
42 outfile = outfile || "aa"
43 for (suffix = outfile; *suffix != 0; suffix++)
45 suffix--
47 *suffix = 'a' - 1
81 if (++*suffix > 'z')
82         *suffix = 'a'
83         ++*(suffix - 1)
87 creat(outfile, 0644)
```

The slicing reduces the 104 line program to 12 lines. As we can see, the program fragment for the generation of filenames is very small even though not trivial compared with other programs we have considered. By symbolic evaluation, and tracing through the loop several times, the result is

```
("x" | argv[*])  || ("aa" | "ab" | ... ).
```

Using German and Wegbreit's methods for the derivation of the loop invariant, we have the conditions *suffix > 'z', *(suffix+1) > 'z', *suffix = 'a' - 1, and *suffix is not decremented in the loop. From these conditions, the following represents possible value for the filenames.

```
("x" | argv[*]) || a || b.
```

where "a" $\leq$ a, b $\leq$ "z".

The user would accept split as safe, as it writes only to files that he expects.

As another example, consider 'cp' which copies files. The synopsis is

"cp filename1 filename2"

or

"cp filename ... dirname".

In the first format, cp copies filename1 to filename2. In the second format, cp copies the filename ... to the directory dirname. The sliced program fragment is like

```
39 creat(argv[2], sbuf.stmode & 0777)
70 ptr = argv[argc - 1]
71 dp = dirname
```

```
72 while (*ptr != 0) *dp++ = *ptr++
74 *dp++ = '/'
75 ptr = argv[i]
78 while (*ptr != 0) ptr++
79 while (ptr > argv[i] && *ptr != '/') ptr--
80 if (*ptr == '/') ptr++
81 while (*ptr != 0) *dp++ = *ptr++
82 *dp++ = 0
84 creat(dirname, sbuf.stmode & 0777)
```

The original program is 154 lines. lines. The sliced fragment is very small and most of the programming statements are strings operations consisting of small loops.

3.4 Limitations of the Filter

Since static analysis techniques are crucial to the operation of the virus filter program, it is assumed that the program being analyzed is constrained to *good* programming practice. That is, the code segment cannot be altered and control may not be transferred to the data segment or the stack segment. These constraints are satisfied for Sun UNIX 3.2 programs. Most programs do not change their code segment or try to execute the data segment. Dynamic linking programs and debuggers are exceptions, albeit important ones. Furthermore, our model assumes a single thread of control. Modifications would be required for a parallel program with shared memory. Some specific limitations of the current virus filter design are discussed below.

3.4.1 Pointer reference

There is no constraint on indirect memory references (any pointer or array reference) in our low-level language. Whenever this kind of reference is associated with a string of unknown length or with a non-deterministic event (e.g. a user input), any memory cell can potentially be modified. The overwritten cell can be anywhere, possibly a filename argument to system calls.

If the symbolic evaluator works conservatively, its output will be too pessimistic, as most storage contains non-determined values. Otherwise the output of the evaluator could be incorrect and a virus could slip into the system without being detected.

The following are common statements in C programs; see Figure 1.

```
while (*p++ = *q++);

--------------------

scanf(''\%d'', &i);
str[i] = 'x';
```

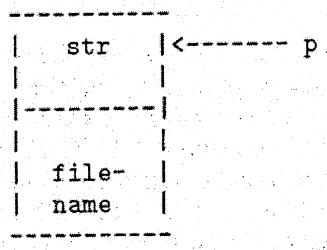


Figure 1. Two examples to illustrate the difficulty of deciding pointer references.

In the first example, if p points initially to a variable 'str' and memory is allocated as shown, p can overflow and, potentially, clobber storage that holds the filename argument to a future system call. We potentially have to determine all possible values of pointers in order to determine what storage can be clobbered.

The undisciplined use of pointers severely hampers effective static analysis, but also represents bad programming style. In the first example, a better alternative is to use

```
strncpy(p, q, MAXLEN).
```

The second example should have a conditional statement such as

```
if ((i < 0) || (i >= UPPER_BOUND)) input_error();
```

before the array reference to avoid an out-of-bound array reference. Assuming these extra statements, it is possible to bound the pointer references to facilitate analysis.

We may be able to infer the range of pointers from the program and prove the pointers are constrained with their associated variable. For example, during the life of a pointer P associated with string S, our filter proves the assertion $\{S \leq P \leq S + \text{length}(S)\}$.

Another difficulty with pointers is with respect to dynamic allocation of memory. It is almost impossible to find the bounds of pointers pointing to dynamically allocated memory variables because there is no way to determine their addresses statically. If we assume the worst case – all pointers can share all dynamically allocated storage – it is not likely that the filter can perform an effective analysis of the program.

Although not acceptable in all situations, it appears that software users can impose a strict programming style on their vendors to simplify the work of the virus filter in pointer analysis.

3.4.2 Loops

As discussed before, the presence of loops makes symbolic evaluation much more complicated because of the indefinite number of iterations. Several methods may suggest solutions to this problem. One of these is to determine the maximum number of iterations of the loop and have the symbolic evaluator go through the loop that number of times. Other methods include the determination of loop invariants to give significant representation for the loop. A method which uses linear inequalities to constrain the variables may be useful [5]. However, all existing solutions are heuristics and do not work in all situations.

3.4.3 Structured Data Types

The symbolic evaluator needs to understand structured data such as strings and records. String operations are common in generating filenames. The evaluator should be able to understand that the code fragment *while (*p++)* is equivalent to moving the character pointer p to the end of the string. Some heuristics are helpful in giving understandable reports to a user. For instance, it is preferable for the filter to output the statement, "A new string S1 is generated from S by appending character 'c' to it" rather than output the assertion

$$\begin{aligned}&\{\ \exists j((\forall i < j : S[i]! = 0 \text{ and } S[j] == 0) \text{ and}\\ &(\forall i < j : S1[i] = S[i]) \text{ and}\\ &S1[j] =' c' \text{ and}\\ &S1[j+1] = 0)\ \}.\end{aligned}$$

In some situations, the complete filenames may not be generable in the absence of specific details of the environment. For example, when a temporary file is generated with the constant prefix */tmp/vi* and the process-id, the symbolic evaluator is unable to give the exact filename because the filename depends on the runtime environment. However, if the evaluator is sufficiently intelligent, it may give a partial result such as */tmp/vi** as the generated filename. However, the cost of having such intelligence is not low as the evaluator needs to understand the semantics of strings and essentially all possible operations on a string.

4 Conclusions and Future Work

There is a need for improved defenses against computer viruses. Defenses include

1. preventing the propagation of viruses
2. detecting an infected program
3. determining if a newly issued program contains a virus.

This paper concentrates on (2) and (3). Our detector tool checks for duplication of services. A program linked with the standard library, typical of UNIX systems, will contain no duplication of system services. A simple virus would carry its own services and would be easily detected by our detector tool. The detector can be defeated, but only by a virus that searches for the services in a program; such a virus will be more complex than current viruses and might contain code that an extension of the detector would flag as malicious.

The filter tool extends the concepts proposed by Crocker and Pozzo. It carries out a static analysis of a given program to determine the capability of a program to modify files. A user of the program under test could then determine if any unexpected files are written to, for example those obtained by searching a directory. The filter uses verification techniques, but since only a subset of a program is usually concerned with filename generation, the technique appears to be more feasible than verification in general. We have simulated the behavior of the filter on typical system programs, such as `date`, `split` and `cp`. Heuristics are required to generate loop invariants (but the loops appear to be quite simple) and to demonstrate that pointers are well-behaved. Implementation is underway.

The prototype of the virus filter will be tested on MINIX system utilities executing on a 80286-based machine. MINIX is a UNIX-like operating system written by Tanenbaum [9]. The MINIX programs are usually small, making them ideal for an initial evaluation of the virus filter. Also, the assembly language is quite simple, which will simplify the translation to the intermediate language.

The similarity between the detector tool and the filter tool is that both attempt to determine if a program under test contains suspicious code. The difference lies in the suspicious code under searching. The detector tool considers program structure, i.e. the way that system calls are made. The filter focuses on the arguments to the system calls. System calls are interesting because a program may interact with other objects in the system, hence cause damage, only through operating system calls.

Generalizations of the detector would involve more complex checks on program structure. For example, the detector might look for the getdirentries (get directory entries) system call which is useful to viruses, but not to most programs. Different compilers generate code in slightly different ways. If the virus code is compiled with a foreign compiler, the detector may be able to detect it with statistical or pattern-matching methods. Furthermore, it is common for a virus to attach itself to the beginning or the end of a program. By looking at the pattern of flow controls, we may obtain some hints to the presence of viruses.

The filter tool uses symbolic evaluation and verification to determine the possible arguments of the system calls. It can be extended to determine values of variables in the program; hence we can prove assertions composed of program variables, which characterize the program behavior. The filter may determine the input conditions which lead to execution of certain sections in the program. For example, if we apply this technique to the *login* program and ask for the condition that the *setuid* statement is executed, the filter should find that a necessary condition is the matching of passwords.

The techniques described in this paper are not limited to the detection of viruses. Trojan Horses are detected in similar way, albeit the detector and filter need to be programmed with different properties.

Because virus detection is undecidable (see [1]), these tools certainly cannot claim to have the ability to detect all viruses. The detector tool can be defeated in at least one sure way by using the existing services of a program. Similarly, the filter can also be defeated: a virus can propagate through the files to which the program has legitimate access. Although these tools cannot detect all viruses, viruses have to hide all the traits we are looking for. A virus designed with these constraints is very complicated, cannot be very infective, and also very hard to write. The mere complexity of such a virus should lead to its early discovery by more common methods of debugging.

5 Acknowledgments

John M. Collins of St. Albans, Herts, England, wrote the disassembler used in Snitch. We are also grateful to Steve Crocker, Maria Pozzo, Doug Mansur and his colleagues at the Lawerence Livermore National Laboratory for many conversations on the virus problem.

References

[1] Fred Cohen. "Computer Viruses: Theory and Experiments", *Computer Security: A Global Challenge*, J.H. Finch and E.G. Dougall (eds.) (1984)

[2] Fred Cohen. "A Cryptographic Checksum for Integrity Protection", *Computers & Security*, Vol. 6 pp. 505-510 (1987)

[3] Fred Cohen. "Models of Practical Defenses Against Computer Viruses", *Computers & Security*, Vol. 8 pp. 149-160 (1989)

[4] Steve Crocker and Maria M. Pozzo. "A Proposal for a Verification-Based Virus Filter", *Proc. of the 1989 IEEE Computer Society Symposium on Security and Privacy*, May 1-3, Oakland, California, pp. 319-324 (1989)

[5] Steven M. German and Ben Wegbreit. "A Synthesizer of Inductive Assertions," *IEEE Trans. on Software Engineering*, Vol. SE-1, No. 1 (1975)

[6] Mark K. Joseph and Algirdas Avizienis. "A Fault Tolerance Approach to Computer Viruses", *Proc. IEEE*, pp.52-58 (1988)

[7] Aamer Mahmood and E. J. McCluskey. "Concurrent Error Detection Using Watchdog Processors–A Survey", *IEEE Transactions on Computers*, Vol. 37, No. 2 pp. 160-174 (1988)

[8] Maria M. Pozzo and Terence E. Gray. "An Approach to Containing Computer Viruses", *Computers & Security*, Vol.6 pp. 321-331 (1987)

[9] Andrew Tanenbaum. *Operating Systems: Design and Implementation*, Englewood Cliffs, N.J., Prentice-Hall, Inc. (1987)

[10] Paul A. Karger, "Limiting the Damage Potential of Discretionary Trojan Horses", *Proc. of the 1987 IEEE Computer Society Symposium on Security and Privacy*, Oakland, California, pp. 32-37 (1987)

[11] D.R. Wichers, D.M. Cook, R.A. Olsson, J. Crossley, P. Kerchen, K.N. Levitt, R. Lo. "PACL's: An Access Control List Approach to Anti-Viral Security", *to appear in Proc. of the National Computer Security Conference, 1990.*

NATIONAL INSTITUTE of STANDARDS and TECHNOLOGY/
NATIONAL COMPUTER SECURITY CENTER

13th National Computer Security Conference

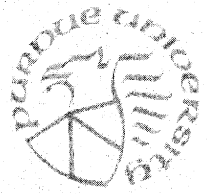
Omni Shoreham Hotel
Washington, D.C.
1 - 4 October, 1990

Proceedings

VOLUME I

Information Systems Security:

Standards - The Key to the Future



Welcome!

The National Computer Security Center (NCSC) and the National Computer Systems Laboratory (NCSL) are pleased to welcome you to the Thirteenth Annual National Computer Security Conference. We believe that the Conference will stimulate a vital and dynamic exchange of information and foster an understanding of emerging technologies.

The theme for this year's conference, "Information Systems Security: Standards -- The Key to the Future," reflects the continuing importance of the broader information systems security issues facing us. At the heart of these issues are two items which will receive special emphasis this week -- Information Systems Security Criteria (and how it affects us) and Education, Training, and Awareness. We are working together, in the Government, Industry, and Academe, in cooperative efforts to improve and expand the state-of-the-art technology to information systems security. This year we are pleased to present a new track by the information security educators. These presentations will provide you with some cost-effective as well as innovative ideas in developing your own on-site information-systems-security education programs. Additionally, we will be presenting an educational program which addresses the automated information security responsibilities. This educational program will refresh us with the perspectives of the past, and will project directions of the future.

We firmly believe that security awareness and responsibility are the cornerstone of any information security program. For our collective success, we ask that you reflect on the ideas and information presented this week; then share this information with your peers, your management, your administration, and your customers. By sharing this information, we will develop a stronger knowledge base for tomorrow's foundations.

JAMES H. BURROWS
Director
National Computer Systems Laboratory

PATRICK R. GALLAGHER, JR.
Director
National Computer Security Center

Conference Referees

Dr. Marshall Abrams	*The MITRE Corporation*
James P. Anderson	*James P. Anderson Company*
Jon Arneson	*National Institute of Standards & Technology*
Devolyn Arnold	*National Computer Security Center*
James Arnold	*National Computer Security Center*
Al Arsenault	*National Computer Security Center*
Victoria Ashby	*The MITRE Corporation*
Elaine Barker	*National Institute of Standards & Technology*
Dr. D. Elliott Bell	*Trusted Information Systems, Inc.*
Greg Bergren	*National Computer Security Center*
James Birch	*Secure Systems, Incorporated*
Earl Boebert	*Secure Computing Technology Corporation*
Dr. Dennis Branstad	*National Institute of Standards & Technology*
Dr. John Campbell	*National Computer Security Center*
Dr. Steve Crocker	*Trusted Information Systems, Inc.*
Dr. Dorothy Denning	*Digital Equipment Corporation*
Donna Dodson	*National Institute of Standards & Technology*
Greg Elkmann	*National Security Agency*
Ellen Flahaven	*National Institute of Standards & Technology*
Dan Gambel	*Grumann Data Systems*
Dain Gary	*Mellon National Bank*
Bill Geer	*National Computer Security Center*
Virgil Gibson	*Grumann Data Systems*
Dennis Gilbert	*National Institute of Standards and Technology*
Irene Gilbert	*National Institute of Standards and Technology*
Dr. Virgil Gligor	*University of Maryland*
Capt James Goldston, USAF	*National Computer Security Center*
Dr. Joshua Guttman	*The MITRE Corporation*
Dr. Grace Hammonds	*AGCS, Inc.*
Douglas Hardie	*Unisys Corporation*
Ronda Henning	*Harris Corporation*
Jack Holleran	*National Computer Security Center*
Jim Houser	*National Computer Security Center*
Russ Housley	*XEROX*
Dr. Dale Johnson	*The MITRE Corporation*
Carole Jordan	*Defense Investigative Service*
Sharon Keller	*National Institute of Standards & Technology*
Leslee LaFountain	*National Computer Security Center*
Steve LaFountain	*National Computer Security Center*
Paul Lambert	*Motorola GEG*
Carl Landwehr	*Naval Research Laboratory*
Robert Lau	*National Computer Security Center*

Dr. Theodore Lee	*Trusted Information Systems, Inc.*
Nina Lewis	*University of California, Santa Barbara*
Steve Lipner	*Digital Equipment Corporation*
Terry Losonsky	*National Security Agency*
Dr. Vic Maconachy	*National Security Agency*
Barbara Mayer	*Trusted Information Systems, Inc.*
Frank Mayer	*Sparta*
Vin McLellan	*Boston University*
Catherine A. Meadows	*Naval Research Laboratory*
Dr. Jonathan Millen	*The MITRE Corporation*
William H Murray	*Independent Consultant*
Eugene Myers	*National Computer Security Center*
Ruth Nelson	*GTE*
Dr. Peter Neumann	*SRI International*
Steven Padilla	*Trusted Information Systems, Inc.*
Nick Pantiuk	*Grumann Data Systems*
Donn Parker	*SRI International*
Rich Petthia	*Software Engineering Institute*
Dr. Charles Pfleeger	*Trusted Information Systems, Inc.*
Jerrold Powell	*Department of the Treasury*
Maria Pozzo	*University of California, Los Angeles*
Michael Rinick	*Central Intelligence Agency*
Ken Rowe	*National Computer Security Center*
Prof Ravi Sandhu	*George Mason University*
Marv Schaefer	*Trusted Information Systems, Inc.*
Dr. Roger Schell	*GEMINI*
Dan Schnackenberg	*Boeing Aerospace*
Miles Smid	*National Institute of Standards & Technology*
Suzanne Smith	*Los Alamos National Laboratory*
Brian Snow	*National Security Agency*
Prof. Gene Spafford	*Purdue University*
Dr. Dennis Steinauer	*National Institute of Standards & Technology*
Freddie Stewart	*ANSER*
Dr. Cliff Stoll	*Harvard - Smithsonian Center for Astrophysics*
Marianne Swanson	*National Institute of Standards & Technology*
Mario Tinto	*National Computer Security Center*
Ann Todd	*National Institute of Standards & Technology*
Eugene Troy	*National Institute of Standards & Technology*
LTC Ray Vaughn, USA	*National Computer Security Center*
Grant Wagner	*National Computer Security Center*
Jill Walsh	*INCO, Inc.*
Wayne Weingaertner	*National Computer Security Center*
Roger Westman	*INCO, Inc.*
Roy Wood	*National Computer Security Center*

Thirteenth National Computer Security Conference
October 1-4, 1990
Washington, DC

Table of Contents

VOLUME I

TRACK B - Systems

VOLUME 2

TRACK C - I - Management & Administration