A Model for Detecting the Existence of Unknown Computer Viruses in Real-Time

Jeffrey M. Voas
Reliable Software Technologies
Penthouse Suite
1001 N. Highland Street
Arlington, VA 22210 USA

Jeffery E. Payne
Reliable Software Technologies
Penthouse Suite
1001 N. Highland Street
Arlington, VA 22210 USA

Dr. Frederick B. Cohen
ASP
PO Box 81270
Pittsburgh, PA 15217 USA

Search terms: Computer viruses, Trojan horses, Models of computation, Virus detection and prevention.

Abstract

This paper describes a model for detecting the existence of computer viruses in real-time.

1 Background

Protection technologies in common use [5] are capable of preventing corruption by viruses (e.g. through mandatory access control), detecting known viruses (e.g. by searching for them), detecting specific types of corruption as it occurs (e.g. trapping the modification of executable files in certain ways), and detecting corruption before it causes significant damage (e.g. through cryptographic checksums in integrity shells), but some points must be made concerning their limited capabilities.

- Any system that relies solely on access control to prevent corruption is no more secure than the individuals or policies allow it to be. Even though sound access control policies against viruses exist, the vast majority of current systems do not implement these controls, and even if these controls are in place, current implementations are imperfect. Even the most sound access control techniques only limit the extent of transitive spread. [2]
- The use of known virus detection schemes doesn't detect viruses unknown to the writer of the defense. New viruses cannot be detected with this technique, and the time and space required for detection grows with the number of viruses.
- Trapping attempts to modify executable files and other similar techniques don't detect viruses corrupting non 'executable' forms, corruption through the use of unusual input parameters, or corruptions in forms other than those sought. They also prevent normal system development activity under some circumstances and are thus of only limited applicability. [5]
- In order for integrity shells to be ideal in an environment with change, we must have some measure of the legitimacy of change. This cannot be effective without some understanding of intent. [3]

Two of these virus detection techniques are *posteriori* in that they are ineffective until after a corruption occurs. In the case of an integrity shell, detection occurs after 'primary' infection, and is capable of preventing 'secondary' infection. In the case of known virus detection techniques, detection doesn't normally occur until some human has detected damage, traced it to a cause, and designed a method for detecting the cause. The a-priori technique of trapping unusual modification behavior is quite weak in that it covers only a small portion of the possible viruses. Access control techniques don't detect viruses, but only attempt to limit their transitive spread.

Several attempts have been made to find techniques which will reliably differentiate programs containing viruses from other programs by examination, dispite the widely know result that this problem is undecidable [1]. Syntactic analysis has been attempted by several authors [9] [11] [16] [17] but none of these have a sound theoretical basis and all result in infinite numbers of false positives and false negatives. In general, the problem of virus

detection is undecidable, and as we would therefore expect, in the most comprehensive study published to date, a 50% false positive and 50% false negative rate was demonstrated [9]. The use of evolutionary and self-encrypting viruses clearly demonstrates the futility of syntactic analysis [4], and virus attackers are now commonly applying this technique.

An alternative approach is to try to differentiate between 'legitimate' and 'illegitimate' behavior by somehow modeling intent. One way to do this is by building up a set of expectations about the behavior of programs and detecting deviations from those expectations. A closely related research result that may be helpful in understanding the present work is 'roving emulation' [6] which has been proposed and simulated as a technique to detect faults at run-time. Results on error latency have shown that effective protection can be attained with sufficient roll-back capability. Other antivirus researchers have also used heuristic approaches for run-time detection of virus-like activities [18] with substantial success.

In the remainder of this paper, we shall consider the concept of detecting behavioral changes in some more depth. We begin by providing a mathematical model of program execution and describing the difference between legitimate and illegitimate program behavior based on behavioral expectations. Next we describe some of the difficulties in applying this model to a practical situation by showing the complexity associated with simplistic application of the model and some of the risks associated with less accurate implementations which eliminate complexity in exchange for accuracy. We then describe some lines of research that seem to have hope for providing adequate accuracy while maintaining reasonable complexity. Finally, we summarize results, draw conclusions, and propose further work.

2 Some Formalities

We commonly refer to computer programs, and by that term, we intend to indicate a transformation of the form [15]:

$$P{:=}(I, O, S, f : I \times S \rightarrow (S^+, O))$$

where:

$\mathcal{I} = \{1, \ldots, \infty\}$	The integers
$I = \{i_0, \ldots, i_m\}, m \in \mathcal{I}$	The input symbols
$O = \{o_0, \ldots, o_n\}, n \in \mathcal{I}$	The output symbols
$S = S^+ = \{s_0, \ldots, s_p\}, p \in \mathcal{I}$	The internal (next) states

We also define I^* as the set of all possible input sequences (the Kleene closure of I) and the 'histories' H of a Moore machine as the output and state sequences generated by applying I^* to the machine from each possible initial state.

A program executing in an undesirable manner, by definition, has some undesired execution behavior, in that it produces undesirable outputs O or 'next states' S^+. Since internal program states may be observed during execution by program instrumentation, and dynamic techniques exist to observe the propagation of program state errors [8, 13], we may be able to create similar techniques to detect certain types of undesirable behavior.

Using the Moore model of computation, we could differentiate 'expected' and 'unexpected' behavior by instrumenting the state-space of a program as follows:

> Before a virus modifies a program 'P_x' in storage, P_x has some functionality f_x. If a modified form of P_x acts differently from the original, we have new program P_y with functionality f_y, where $f_x \neq f_y$ for at least one input sequence ($\mathbf{I} \in I^*$). As P_y executes, and $\mathbf{I}$ is provided as the input sequence, f_y must produce an internal state s_y and/or output sequence $\mathbf{O}_y$ that differs from those of P_y under input $\mathbf{I}$ (i.e. $s_y \neq s_x$ or $\mathbf{O}_y \neq \mathbf{O}_x$).

Similarly, a 'parameter altering' corruption may corrupt the behavior of a program P_x which normally operates on a subset D_x of all the possible inputs ($D_x \subset I^*$) by forcing it to operate on a different subset (D_y) of inputs.

> When some input sequence in a set D_y is provided to P_x, we observe a modified history denoted by H_{D_y}, where $H_{D_x} \neq H_{D_y}$ and $\nexists d \in D_x, d \in D_y$. Any differences between H_{D_x} and H_{D_y} signal an unanticipated application of P_x.

The analogy between corruptions caused by computer viruses or other malicious attacks and program state errors caused by random faults or mistakes is the basis of this paper. In this model, we detect corruption in real-time before the corruption propagates. This has several advantages in non-viral attacks as well. For example, a common method for bypassing access control is by providing unexpected input parameters to operating system calls in the hope that the implementation fails to trap some condition and grants an inappropriate access. Trojan horses pose similar problems in that they perform some unanticipated function under some conditions. The same technique offers hope for detecting Trojan horses when they become active and thereby preventing some of their effects.

In this model, we view an executing program as a series of dynamic state spaces that are constantly being modified, instead of the conventional view of programs from as static syntactic spaces. This is quite different from the previous efforts in virus detection, and there is therefore some hope that this model will provide the basis for some form of rapid real-time detection of unanticipated system activities. The problem with this model is that for all but the most trivial of examples, the size of the state-space is enormous. The issue that remains to be resolved is how to get the practical advantages of this technique without the enormous time and space penalties inherent in its use.

3 Discussion

The model discussed here is of a deterministic Von Neumman computer in which a stream of instructions is executed in order to effect a required transformation of the input to the output.

We consider the *program state* of the computer at a point in execution to include all of the information necessary to restart the computation if the executing program is interrupted. In a synchronous finite state machine model of computation, the state is specified as the memory state of all of the memory registers of the machine and the current clock value (high or low).

In practice, the state of a machine may not be that simple. In a timesharing system with good process protection, the program state typically includes the state of all process specific processor registers and all of the internal memory of the program. In a personal computer without good protection, the state of a program may be far more complex, but is almost always encoded as the entire memory and register state of the machine. For a more accurate restart of a process, many other factors may be involved, including the state of input and output ports and buffers, the disk state of the machine, the state of any mounted devices, etc. We model all relevant state information as a set of variable/value pairings. State information is presented as a finite set $\{p_1, p_2, \ldots, p_n\}$ where each p_i is an ordered pair (identifier, value), and where initial values may be indeterminate. For example:

$$\{\text{(a, 5), (b, 1), (c, 300.2), (d, undefined), (pc, 3)}\}$$

might be part of the program state of a program with the four variables 'a', 'b', 'c', anc 'd', immediately after executing statement 2 ('pc' in this case indicates the 'program counter' register). The variable a has the value 5, b has the value 1, and c has the value 300.2.

The use of *undefined* means that at some point in the execution of the program the value of d became indeterminate (i.e. no assertion can be made about its value at this point). If we know only about the high level semantics of a program and do not wish to make assertions about how the processor interprets instructions, an example of an indeterminate value would be the value of a pointer after its storage has been freed.

For convenience, we may assume that all of the values associated with a given identifier are of the same type. This is reasonable in the general case because we can assign each bit of program state to a unique identifier, which will effectively make them the same type. The *program state domain*, $\mathcal{D}$, is constructed by taking all possible values of each component of the program state at all instructions. To derive the history for any particular program execution under any given input sequence, we can characterize each instruction in the executable program in terms of its effect on the program state.

There are many methods for doing this, and for most modern computers, we can do this very easily through the use of a hardware definition language and a simulator. For any single input sequence, this is not particularly difficult, since in the worst case, we require only one copy of the machine state for each instruction executed, and in most cases, each instruction execution is simulatable in a fixed time. Thus the time and space required is at most linear with the number of instructions executed.

The complexity problems start when we wish to characterize a large number of different input sequences. For each input symbol, we have as many possible program executions as

there are symbols in I, and in general, for an n-step program execution sequence, we have $|I|^n$ possible executions. Assuming there are m bits of state information and that input symbols require k bits to represent, we require $m(2^{k^n})$ bits of state information. For any non-trivial input set and program, this is enormous. For example, for a program with only 1 byte of state information and inputs of at most 1 byte each executing only 50 instructions requires over 1 googol (10^{100}) bytes of storage to characterize! [1]

In a real computer, external inputs can come at any of a large number of different times, and the normal processing mechanism allows 'interrupts' which alter program execution sequences between, or in rare cases during, instruction execution. Multiprocessing introduces further uncertainties, particularly in DOS based computing environments where interprocess interaction is truly arbitrary and uncontrolled. We simply cannot count on anything about programs in these environments. Keeping the complexity issue in mind, we will press on in the hope that we may eventually be able to collapse these very large numbers into manageable and effective protection mechanisms.

4 Reduced Complexity Models

One way to collapse much of the state information associated with a program is to assume that there is a separation between 'program' and 'data'.

Let P consist of the set of m transitions [2] $T = \{t_1, \ldots, t_m\}$. Let $\mathcal{A}_{t_j,P,r,x}$ represent the program state that exists after executing instruction t_j on the r^{th} iteration of t_j by input x from $domain(P)$, where $domain(P)$ represents all 'valid'input sequences to program P. There may be other inputs on which P could execute that would cause undesired states. Let n_{x,t_j} represent the number of times that instruction i_j is executed by input x. Formally, $\mathcal{D}$ is:

$$\bigcup_{k=1}^{m} \bigcup_{x \in domain(P)} \bigcup_{r=1}^{n_{x,t_k}} \mathcal{A}_{t_k,P,r,x}.$$

The execution of a single program instruction may, in general, change any number of components of the data state, resulting in an entirely new data state, but in practice, the vast majority of transitions alter only a very small portion of the data state. As a sequence of transitions in a program is executed by the computer, the initial program state is successively transformed until the program halts or loops indefinitely.

$\mathcal{D}$ represents the expected internal state behavior of P (i.e. at any snapshot into the execution of P, we should find the program state of P is identical to some program state in $\mathcal{D}$). If this does not occur, then P has created a program state that is not possible according

[1] $(2^8)^{50} = 256^{50} > 10^{100}$

[2] a.k.a. instructions

to the $domain(P)$. This signals that P is in a state that cannot be achieved by any input in $domain(P)$ and suggests that:

1. P has received an input that is not in $domain(P)$, or
2. P has been altered by malicious code, or
3. The execution of P has not proceeded according to the model.

If P is receiving invalid inputs, it is not necessarily the case that a security violation has occurred, however it is a situation that may require attention. It could be that some sensor that sends inputs to P has failed. Or it could be that malicious code is sending perverse input parameters to P. If the behavior of P has been altered, then we have a warning that potentially malicious code has affected the internal behavior of P. A hardware failure, modeling error, or implementation inaccuracy could also result in an equivalent result.

To assure that the program states that are being created are not being influenced by malicious code, each program state created during execution needs to be checked against the members of $\mathcal{D}$. A theoretical algorithm for producing a warning that a program state has been created that is not in $\mathcal{D}$ follows:

1. Create the set $\mathcal{D}_{t_k} =$

$$\bigcup_{x \in domain(P)} \bigcup_{r=1}^{n_{x,t_k}} \mathcal{A}_{t_k,P,r,x}$$

   for each instruction t_k in T. $\mathcal{D}_{t_k}$ contains every program state that could ever occur after transition t_k, given that the program input is in $domain(P)$.

2. During the execution of P in its operational environment, insert probes after each transition t_k to sample the program states being created. Determine whether the sampled program states are in $\mathcal{D}_{t_k}$. If they are not, produce a warning.

5 Building a Practical Virus Warning System

The cardinality of $\mathcal{D}$ is enormous, so it is infeasible to create and store every $\mathcal{D}_{t_k}$. Even if $\mathcal{D}$ were not so large, there may exist an input x for which n_{x,t_k} is so large that $\mathcal{D}_{t_k}$ cannot be stored. This means that the theoretical algorithm is generally impractical, however there are several ways in which the algorithm can be partially implemented. Since we are unable to fully implement the algorithm, we must accept a risk of false negatives.

To partially implement the theoretical model, we have many options. We could, for example, store a small random sample of D at a random sampling of instructions; but

this would yield enormous numbers of false positives, since we could never hope to store a substantial portion of D at any place, and thus the likelihood of an uncovered state would be very high. A more directed approach might be to select the portion of the state to be stored and places in the program at which to store and compare so that we know that certain things are expected to be the case. This mechanism is provided in some computer languages, in which invariants are specified by the programmer for the purpose of automating certain aspects of program proofs [14].

Another approach is to limit ourselves to particular classes of attacks. For example, we could identify specific instructions that computer viruses locate to attack executable code and select those locations for performing tests. By assuming that each transition t_k in a target program does not have an equally likely chance of being attacked, we can reduce the search space. Another approach might be to use information content measures to identify instructions or state space characteristics that are more or less likely to occur in the program being analyzed, and identify executions wherein these characteristics are not followed for an identifiable portion of the program execution.

The latter approach is particularly nice since we have to store only a very small amount of data state information and evaluate a relatively simple metric in order to make a determination. Of course, the number of false positives and false negatives will depend heavily on how tight the bounds of the program execution characteristics are, but then it is also quite simple to alter the bounds by simply changing the metric for different applications.

If we are looking for particular types of attacks, we might also try to do a variant on fault-tree analysis [7, 10]. We first determine a set of unacceptable output states (i.e. enumerate disastrous output states) and then we apply a dynamic technique termed propagation analysis [8, 12, 13] to determine where program states can be created in the program text that could result in the disastrous output states that we enumerated. The key to making propagation analysis effective is the ability to simulate the internal behavior of viruses. As an example, in a timesharing system we could identify places in the program where system calls are made as a place where disastrous output could originate.

We do not pretend that this is trivial, but it can be made a lot simpler through additional effort in the specification and requirements phases during software development. In a sense, this is a preliminary "design-for-security" step. Applying propagation analysis provides a list of source locations in which a disastrous internal program state could be created that could propagate producing undesirable outputs. We then map the source locations to the object instructions (i.e. find the instructions that execute the source locations identified as dangerous). Next we add instrumentation instructions to generate samplings for $\mathcal{D}_{t_k}$s. During normal execution, we place self-test instructions at these locations to detect variations.

6 Summary, Conclusions, and Further Work

We have briefly explored the possibility of using experimental analysis of program behavior to differentiate between legitimate and illegitimate program execution, described some of the complexity problems associated with this approach, and contemplated the possibility of reducing that complexity to a reasonable level.

The general approach of detecting the intrusion by analyzing state information in executing programs has the appeal that it could result in earlier detection than can be attained through other general purpose techniques, while providing more generality than attack specific defenses. The concept lends itself to detecting a very broad range of attacks including Trojan horses and viruses, attacks generated by unanticipated input sequences, and even unintentional corruptions resulting from transient or permanent faults.

This research is still in a very early stage, and clearly we can only draw limited conclusions. Although the general problem of virus detection is undecidable, the problem of characterizing known program behavior in a finite state environment is *only* exponential in the number of instructions executed. Furthermore, the upper bound on state sequences is far higher than we would expect in normal program execution and complete state transition information may not be required to detect many program variations. The possibility of a practical defense based on this idea is therefore still an open question.

A great deal of further work will be required to determine the feasibility of this line of defense in practical environments. More specifically, tighter bounds on the complexity of real program state spaces for particular classes of programs should be determined, and there is a very real possibility that this will yield viable partial solutions. Several ideas have been raised and analysis of these ideas may yield useful results, particularly for special classes of programs such as those written in certain languages or those that compute particular sorts of functions. There is also the possibility that programs generated from mathematical specifications may provide particular analytical advantages in detecting improper execution and that programs written with additional protection related information may yield far more efficient defensive techniques based on this concept.

References

[1] F. Cohen. Computational Aspects of Computer Viruses. *Computers & Security*, 8:325–344, 1989.

[2] F. Cohen. Protection and Administration of Information Networks with Partial Orderings. *Computers & Security*, 6:118–128, 1987.

[3] F. Cohen. Models of Practical Defenses Against Computer Viruses. *Computers & Security*, 8:149–160, 1989.

[4] F. Cohen. A Short Course on Computer Viruses. *ASP Press* Pittsburgh, PA USA, 1991.

[5] F. Cohen. Defense-In-Depth Against Computer Viruses. *Computers & Security*, awaiting publication, 1992

[6] M. Breuer, F. Cohen, and A. Ismaeel. Roving Emulation. In *Proceedings of Built-in Self-test Conf.*, March 1983.

[7] E. Henley and H. Kumamoto. *Reliability Engineering and Risk Assessment.* Princeton-Hall, Inc., Englewood Cliffs, N.J., 1981.

[8] J. Voas, L. Morell, and K. Miller. Predicting Where Faults Can Hide From Testing. *IEEE Software*, 8(2), March 1991.

[9] M. Pozzo. PhD thesis, U. of California, Los Angeles, 1990.

[10] R. Barlow, J. Fussell, and N. Singpurwalla. *Reliability and Fault Tree Analysis: Theoretical and Applied Aspects of System Reliability and Safety Assessment.* Society for Industrial and Applied Mathematics, 1975.

[11] F. Skulason. The 'F-prot' virus detection program uses several unpublished examples of these techniques.

[12] J. Voas. A Dynamic Technique for Predicting Where Data State Errors May be Created that Cause Certain Classes of Software Failures. Submitted 1992.

[13] J. Voas. A Dynamic Failure Model for Estimating the Impact that a Program Location has on the Program. In *Lecture Notes in Computer Science: Proc. of the 3rd European Software Engineering Conf.*, volume 550, pages 308–331, Milan, Italy, October 1991. Springer-Verlag.

[14] D. Wortman. On legality assertions in EUCLID. *IEEE Trans. on Software Engineering*, SE-5(4):359–366, July 1979.

[15] E. Moore Gedanken Experiments on Sequential Machines in Automata Studies, C. Shannon and J. McCarthy, ed. Princeton University Press, Princeton, NJ, pp129-153, 1956.

[16] P. Kerchen, R. Lo, J. Crossley, G. Elkinbard, K. Levitt, and R. Olsson Static Analysis Virus Detection Tools for Unix Systems. 13th National Computer Security Conference, Washington, D.C., USA October, 1990

[17] M. King Identifying and Controlling Undesirable Program Behaviors 14th National Computer Security Conference, Washington, D.C., USA October, 1991

[18] S. Chang, et. al. Internal documents and discussions with personnel at Trend Microdevices regarding their commercial antivirus product, 1990.