Computers & Security, 12 (1993) 275–283

A model for detecting the existence of software corruption in real time

Jeffrey M. Voas[1], Jeffery E. Payne[1] and Frederick B. Cohen[2]

[1] *Reliable Software Technologies, Penthouse Suite, 1001 N. Highland Street, Arlington, VA 22210, USA*
[2] *ASP, PO Box 81270, Pittsburgh, PA 15217, USA*

This paper describes a model for detecting the existence of software corruptions in real time.

Keywords: Integrity protection, Models of computation.

1. Background

Protection technologies in common use [8] 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 they occur (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 [4].

- The use of known virus detection schemes does not 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 do not 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 [8].

• 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 [5].

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 does not 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 do not 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, despite the widely known result that this problem is undecidable [6]. Syntactic analysis has been attempted by several authors [10–14] 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 [13]. The use of evolutionary and self-encrypting viruses clearly demonstrates the futility of syntactic analysis [7], 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" [2] 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 [3], 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 [12]

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

where

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

We also define I^* as the set of all possible input sequences (the Kleene closure of I), O^* and S^* as the Kleene closures respectively of O and S, H as a single "history" of a machine (i.e. a valid set of $\{I^n, O^n, S^n\} \in f$), H^* as the set of all "histories" of the machine (i.e. all possible H for a particular machine), and H^A as the set of all histories of a machine under inputs in A. Subscripts of these variables are used to denote specific machines.

A program executing in an undesirable manner, by definition, has some undesired history. Since internal program states may be observed during execution by program instrumentation [15, 17], we may be able to detect undesired histories by detecting unexpected states or outputs. Under this assumption, we might be able to differentiate "expected" and "unexpected" behavior by instrumenting the "state-space" of a program as follows:

Before a program "P_x" is modified, P_x has some functionality f_x. If a modified form of P_x acts differently from the original, we have a 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 a history H_y that differs from H_x under input $\mathbf{I}$. More explicitly

$$\exists \mathbf{I} \in I^*, f_x \neq f_y \rightarrow H_x \neq H_y$$

A "parameter altering" corruption may alter the expected operation of program P_x which is supposed to operate 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 improper input sequence "d" ($d \in D_y$, $d \notin D_x$) is provided to P_x, we observe a history H_x^d, where $H_x^d \notin H_x^{D_x}$. Any differences between $H_x^{D_x}$ and $H_x^{D_y}$ signal an unanticipated application of P_x.

Note that a similar case can be made for subsets of O^*, since the external behavior of two equivalent machines is equivalent under our model, regardless of whether their internal states are equivalent or similarly coded. This case cannot, however, be made for S^n, because in many machines, it is possible to produce a multitude of output sequences using identical subsets of internal states. The simplest example is a 1-state machine in which input is simply sent to output. In this case, any output sequence can be generated with an identical state sequence. More complex machines have similar properties.

The analogy between corruptions caused by computer viruses or other malicious attacks and program state errors caused by random faults should be clear. In this model, we detect corruption in real time before the corruption propagates. This has the potential for very broad application. 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, thereby preventing some of their effects.

In this model, we view an executing program as a set of states that are constantly being modified, as opposed to the view of programs as static syntactic entities. This is somewhat 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. There are two major problems with applying this model to non-trivial examples. The first major problem is that if we cannot adequately protect a program from corruption, we may not be able to protect the characterization of valid histories. In this case, we are simply providing systematic redundancy that could potentially be corrupted by an attacker along with the information it is intended to cover. The second major problem is that the size of the state-space is enormous. The complexity 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 *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 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.

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. 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 machine step, 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, a program with only 1 byte of state information and inputs of 1 byte each executing only 50 instructions requires over 1 googol (10^{100}) bytes of storage to characterize!*

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". 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 values may be indeterminate. For example

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

*$(2^8)^{50} = 256^{50} > 10^{100}$.

might be part of the program state of a program with the four variables "a", "b", "c", and "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. The *program state domain* $\mathscr{D}$ is constructed by taking all possible values of each component of the program state at all instructions.

Let P_x consist of the set of "transitions"* $T_x=\{t_1,\ldots, t_m\}$, $m\in\mathscr{I}$. Let $\mathscr{A}_{t_j,P_x,r,n}$† represent the program state that exists after executing instruction t_j on the rth iteration of t_j by input d from D_x. There may be other inputs on which P could execute that would cause undesired states. Let c_{n,t_j} represent the "count" of times that transition t_j is executed by input n. Formally, $\mathscr{D}$ is

$$\bigcup_{k=1}^{m}\bigcup_{n\in D_x}\bigcup_{r=1}^{c_{n,t_k}}\mathscr{A}_{t_k,P_x,r,n}$$

The first union covers each instruction, the second union covers each input of each instruction, and the third union covers each iteration of each instruction on each input.

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.

$\mathscr{D}$ represents the expected internal state behavior of P_x (i.e. at any snapshot into the execution of P_x, we should find the program state of P_x is identical to some program state in $\mathscr{D}$). If this does not occur, then we have encountered a program state that is not possible from P_x operating on D_x. This signals that either

(1) P_x has received an input that is not in D_x, or

(2) the program being executed is not P_x, or

(3) the execution of P_x has not proceeded according to the model.

If P_x 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_x has failed. Or it could be that malicious code is sending perverse input parameters to P_x. If the behaviour of the program has been altered, then we have a warning that potentially malicious code has affected the internal behavior of P_x. A hardware failure, modeling error, or implementation inaccuracy could also result in an equivalent result.

To ensure 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 $\mathscr{D}$. A theoretical algorithm for producing a warning that a program state has been created that is not in $\mathscr{D}$ follows:

(1) Create the set $\mathscr{D}_{t_k}=$

$$\bigcup_{n\in D_x}\bigcup_{r=1}^{c_{n,t_k}}\mathscr{A}_{t_k,P_x,r,n}$$

*Also known as instructions.
†The state "after" the instruction is executed.

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

(2) During the execution of P_x 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 $\mathscr{D}_{t_k}$. If they are not, produce a warning.

In most modern computers, low-level program states cannot effect all of memory. In fact, for most programs on most computers, few if any instructions impact more than a few bytes of storage. This has a dramatic impact on the size of $\mathscr{D}$. For example, let us assume that the incoming distribution of value for x is uniform over all integers and look at a sample program "P".

Program P_1:=

```
read(x);          data space 1
x=x+1;            data space 2
x=x*2;            data space 3
x=x/17|256;       data space 4
write(x);
```

Assume that x is a 32-bit integer, so that 4 bytes are needed to store it. Also, assume that we are only concerned with the variable x in our data state and ignore the details of how the sequence is implemented. If we observe P_1 executing with each of 1 000 000 different values of x, storing $\mathscr{A}_{t_1,P_1,I,x}$ (i.e. data space 1) would require 32 000 000 bits.

To store $A_{t_2,P_1,I,x}$ as well, we need relatively less additional memory, because most additions of 1 to x only affect a few bits of the result. For instance, assume that we have the following sequence for one instance of data space 1:

00000000000000000000000000000001

All we really need to store for data space 2 is the low-order two bits of the result, since these are the only things that will change under normal execution. In data space 2, we will have only "10". The storage requirement in this case will be different for each input value. Unfortunately, this provides very poor coverage. Since 1 in 4 values for the 32-bit quantity will end in "10", only 75% of random invalid inputs will be detected by this stored value. On the other hand, if we make the general assumption that any transition t_k alters at most 2 bytes of storage, we can have far better coverage for a relatively small amount of memory. For example, assume that we have

01100001110000011100011100011010

in data space 1. In data space 2, we need only store

00011011

because the high-order 3 bytes are unchanged by the increment transition. Thus instead of needing 4 megabytes in each data space, we will reduce that amount to 2 megabytes. Thus this simple program would require 8 megabytes, 2 megabytes for each data space, for a prior observation period of 1 000 000 executions. As the number of prior observations decreases, the memory requirement decreases, although the potential for a greater number of false warnings also increases as the amount of prior state information is diminished. This is a trade-off between space and accuracy in this model.

Using the same example, suppose a corruption altered the order of execution so that these instructions did not occur at the same step in program execution, and we stored our data spaces as a function of how many instructions have been executed. Since data space 4 leaves all of the high order bytes as 0s, only 1 in 16 777 216 (2^{24}) instructions that impact x will produce proper results, regardless of the value of x in data space 3! If the attacker alters program execution, the chance of "accidentally" producing a valid alteration of x is certainly minute. This would seem to indicate that by storing a very small number of properly selected

properties of programs during execution, we can detect unexpected behavior while producing a very low rate of false negatives and no false positives.

5. Building a practical virus warning system

The cardinality of $\mathscr{D}$ is enormous, so it is infeasible to create and store every $\mathscr{D}_{t_k}$. Even if $\mathscr{D}$ were not so large, there may exist an input for which c_{n,t_k} is so large that $\mathscr{D}_{t_k}$ 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 implement the algorithm fully, we must accept a risk of false negatives.

To implement the theoretical model partially, we have many options. We could, for example, store a small random sample of $\mathscr{D}$ 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 $\mathscr{D}$ 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 [17].

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 [1, 9]. 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 [15, 16] 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 $\mathscr{D}_{t_k}$. 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] 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.

[2] M. Breuer, F. Cohen and A. Ismaeel, Roving emulation, In *Proc. Built-in Self-test Conf., March 1983.*

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

[4] F. Cohen, Protection and administration of information networks with partial orderings, *Comput. Secur.*, 6 (1987) 118–128.

[5] F. Cohen, Models of practical defenses against computer viruses, *Comput. Secur.*, 8 (1989) 149–160.

[6] F. Cohen, Computational aspects of computer viruses, *Comput. Secur.*, 8 (1989) 325–344.

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

[8] F. Cohen, Defense-in-depth against computer viruses, *Comput. Secur.*, 11 (1992) 563–579.

[9] E. Henley and H. Kumamoto, *Reliability Engineering and Risk Assessment*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[10] 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 Conf., Washington, DC, October 1990.*

[11] M. King, Identifying and controlling undesirable program behaviors, *14th National Computer Security Conf., Washington, DC, October 1991.*

[12] E. Moore, Gedanken experiments on sequential machines, in Automata studies, C. Shannon and J. McCarthy (eds.), Princeton University Press, Princeton, NJ, 1956, pp. 129–153.

[13] M. Pozzo, *PhD Thesis*, University of California, Los Angeles, 1990.

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

[15] 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. 3rd European Software Engineering Conf., Milan, Italy, October 1991, Springer-Verlag, Vol. 550, pp. 308–331.

[16] J. Voas, L. Morell and K. Miller, Predicting where faults can hide from testing, *IEEE Software*, 8(2) (1991) 41–48.

[17] D. Wortman, On legality assertions in EUCLID, *IEEE Trans. Software Eng.*, SE-5(4) (1979) 359–366.