Comments on a paper by Voas, Payne & Cohen: "A model for detecting the existence of software corruption in real time"

Peter Ladkin*   Harold Thimbleby†

Abstract

We discuss a procedure proposed by Voas, Payne & Cohen [6] for detecting the existence of software corruption in real time. In particular, we discuss problems posed by the concurrent execution of programs. In the cases where the proposed method may work, corruption is unlikely to be a problem; and where corruption by viruses and Trojans are a problem, major problems with the method remain.

Keywords: integrity protection, concurrency, consensus problem

Published as: H. Thimbleby & P. B. Ladkin, "Comments on a Paper by Voas, Payne & Cohen: 'A Model for Detecting the Existence of Software Corruption in Real Time'," *Computers & Security*, **13**(6), pp527–531, 1994.

In their recent paper "A model for detecting the existence of software corruption in real time" Voas *et al.* (writing in this journal) suggest a method of providing an alarm to warn against software corruption in real-time [6]. Their motivation is to detect Trojan Horses and/or computer virus infection. Their method rests on the claim that real programs have, in fact, a finite number of possible states, and therefore in principle are equivalent to a finite-state machine (FSM). They suggest building an equivalent FSM, and running it in parallel with the original program on the same input. If there is a discrepancy between the program and the FSM simulator, they suggest that one may conclude that there is software corruption. They suggest that the FSM and the original program can be comparable in terms of speed. Voas *et al.* invited further commentary on their proposal: "This research is still in a very early stage [. . .] A great deal of

*Department of Computing Science, University of Stirling, STIRLING, FK9 4LA, Scotland: pbl@compsci.stirling.ac.uk

†Department of Psychology, University of Stirling: hwt@compsci.stirling.ac.uk.

further work will be required to determine the feasibility of this line of defense in practical environments." This note accepts their invitation.

A practical point about viruses is as follows. To survive, a computer virus (or a biological virus) must have a large enough pool of susceptible hosts. In the case of computer viruses, this certainly happens given the size of (for example) the DOS population. But DOS and other common operating systems are unsuitable for Voas *et al.*'s proposed approach, as we argue below. If, however, the authors propose modifying the operating system to be amenable to their approach, then viruses and other forms of attack would be less likely, purely on epidemiological grounds. Of course, this is a desirable consequence of their approach, or any other approach that runs on a sparsely-distributed operating system. Such an approach, using a far simpler counter-viral strategy, has been investigated in [5].

While Voas *et al.* achieved one of their goals — to make their readers think about a potential solution to corruption problems — there remain a number of major issues to be addressed before their method can be considered for practical use.

The authors use an argument with roughly the following form:

Premiss 1: Programs are functions from their inputs to their outputs;

Premiss 2: Real programs are finite-state; therefore

Conclusion 1: The I/O behavior of the program is what is important about the program; and therefore

Conclusion 2: The program is equivalent to a finite-state machine.

They suggest that one may build such a finite state machine explicitly in software, and run it in parallel, with an alarm sounding when there is a discrepancy on outputs for a given input.

(1) Many programs nowadays, including those running in operating systems as simple as DOS, are *reactive*. Reactive programs are those that run continuously, accepting input and transforming it, maybe passing it on to other programs. The behavior of many reactive programs are described, not by finite-state machines, but by finite-state transducers, finite-state I/O automata, Büchi automata, or by a number of other abstract machines, whose behavior may not simply be described as a set of input-output pairs.

Furthermore, reactive programs, especially those in an operating system, are usually multiprogrammed, if not actually concurrently running. We are unaware of any successful approach to the description of such systems of programs that describes their behavior simply as a set of input/output values for each program. Reactive programs often communicate information or signals to each other (as in an operating system such as Unix) through a channel or through shared memory. In these cases, where communication is *asynchronous* (the reading process may read at a later time than when the writing process produces the information), even a system of two communicating finite-state transducers may have infinitely many states. So, although each individual component has finitely many states, their interaction does not. Furthermore, if the data passed between these two finite-state devices may be structured, it is common knowledge that the two devices along with their channel acquire Turing-complete computational power [2].

So, one may doubt whether the assumption of finitely many states is valid, and this must itself be clearly distinguished from the assumption that every program may be simulated by a finite-state device.

(2) But, the authors may wish to argue, it is unrealistic to assume a channel of infinite capacity. If a system of processes uses a finite amount of memory, then one can include each process, the states of the channel, the message store, and still have only finitely many states, therefore in principle the system still forms a finite state device.

However, this approach is not without difficulties. Cooperating programs may misbehave in certain unstructured ways, in ways that do not fit this model of encapsulated processes communicating via encapsulated memory or channels.

Consider a pair of privileged programs, as may be found in an operating system kernel. They could overwrite a (bounded) message store in the course of their normal operation, *while nevertheless remaining uncorrupted.* They may be badly programmed, yes — but not necessarily corrupt. If we follow the authors' suggestion, then in order to tell what is happening with the programs we must ascertain that this bad behavior happens *in exactly the same way.* Consider a pair of communicating programs overwriting a buffer, which corrupts some partition used by a completely different program. We must now ascertain that the FSM simulation of the pair corrupts the partition in that same way, and with the same unreadable data. So, to use the authors' method for a pair of communicating reactive programs P_1 and P_2, one must be prepared to examine the behavior of other, logically completely unrelated, programs for evidence not just of corruption, but of corruption identical in every respect. (One then has to decide whether the corruption observed in these other programs is corruption due to a virus, or due rather to the improper but uncorrupted behavior of P_1 and P_2.) If one focusses on building the FSM for P_1 and P_2 alone, including the bounded message store, one fails to capture the behavior that influences the execution of other processes. So the suggestion to build a simulating FSM fails, unless one builds an FSM for the system as a whole, single entity, with a precise description of the memory partitioning. This would seem not to be practical (even for non-standard operating systems).

The finite-state assumption requires more careful justification than the authors have given it to assure the successful application of their method to even simple reactive programs.

(3) The authors do not give sufficient explanation on how one may discover which FSM a program might be input-output equivalent to (if it is). The authors suggest that factoring the states into control states and data states may reduce the complexity of description of the FSM, which is indeed true *when one has the FSM already*. However, it may be that to discover the FSM in the first place may take time proportional to *the total number of states* of that FSM. Unfortunately the total number of states of an FSM may be a huge function of the initial program size. It is currently feasible to generate and check roughly 10^{14} states [3] (when one chooses to simulate communicating state machines by a single global-state FSM) but such a state space can easily be created by trivial programs with fewer than 42 branch points in total. To help cope with the state explosion, there are various techniques, such as the 'supertrace' method, for trading-off the size of the state space against the accuracy of the enumeration of states [4].

Since the interaction of two communicating FSMs with data is Turing-complete, Voas *et al.* may want to suggest that this FSM-discovery process must be factored into discovering the control-state FSM and separately a data-state simulator, whence the discovery process would be tractable in the size of the control-state FSM, at least. If so, there needs to be a construction that demonstrates this feature.

(4) Voas *et al.* seem to claim that two replicated versions of a program are sufficient to detect corruption. Given how hard it is to design and build correct software, and the vast number of states of the replicating FSM, one immediately asks how the authors propose to ensure that any discrepancy is caused by corruption — rather than by the faulty design and construction of the FSM? They are also concerned with speed problems and suggest some complex optimisations, which in themselves increase the difficulty of constructing a correct FSM. How do they suggest we may we move from the observation of a discrepancy to the conclusion that corruption is the cause of it, rather than that our programming of the FSM is at fault? Of course, discovering a discrepancy may be a useful indication of poor software quality, but this decision issue must be settled before concluding that the method suffices for its stated purpose.

(5) In the case of computer virus infection, the authors' method cannot determine with certainty that both programs have not been corrupted. Suppose both programs are corrupted, and yield identical but incorrect output behavior. The authors' method would not detect this corruption. In certain environments, with certain implementations of the method, this situation may be less likely than the case where one program and not the other is corrupted. Equally, in other environments with different implementations of the method, it may be more likely. Since the method is not complete (i.e., there are corruptions it cannot detect), what then exactly is the scope of the authors' method?

(6) One reason why viruses and Trojan Horses are a practical problem is that users do not know in advance what programs they wish to run. If the collection of programs they wish to run was a fixed and known set, as for example in embedded systems, then these programs and all software related to them could be consigned to hardware, and they would no longer be susceptible to the sort of interference that concern the authors. Indeed, this technique is used in many embedded systems, such as flight control computers. If, on the other hand, users wish to run programs taken from a large (and generally non-local) repertoire of software, then in order to use the authors' method, either a general procedure is needed for translating (reactive and interacting) programs written in some Turing-complete language (such as C or Pascal) into FSMs, or each program must be translated on an *ad hoc* basis. Given that it is regarded in the industry as currently infeasible to determine the exact I/O relations of related programs in a large set, one wonders how the authors may conclude that this is practical in their case?

(7) Finally, there is a large literature on the basic problem that the authors will need to solve to have a workable method, namely the Consensus Problem, otherwise know as the Byzantine Generals Problem (see [1] for a survey).

The Consensus Problem may be briefly stated as follows. Suppose one has a number of concurrently running processes that compare their output for identity. If there are discrepancies: how do you tell which is at fault; whether the fault is in the process output or the discrepancy-testing-and-reporting algorithm; whether the discrepancy-reporting algorithm itself is accurate, and reports no discrepancy when there is one, or reports one when there is none? Algorithms that solve these problems are presently cumbersome and impractical: this complexity is inherent in many cases in the problem itself.

Voas *et al.* assume the identity test between two programs is a trusted piece of software: it certainly cannot be when it is implemented *in* a system some components of which may themselves be suspect (consider the earlier example of two badly-programmed but uncorrupted processes trashing the store — the store they trash might be associated with the discrepancy-reporting algorithm).

Besides, *two* voting processes, as in the authors' method, is not a number that permits any solution to a non-trivial Byzantine agreement problem, therefore the authors cannot hope to elaborate their method to solve the correctness problems that we have raised. One needs more processes.

References

[1] M. Barborak, M. Malek & A. Dahbura, "The Consensus Problem in Fault-Tolerant Computing," *ACM Computing Surveys*, **25**(2), pp. 171–220, 1993.

[2] D. Brand & P. Zafiropulo, "On Communicating Finite-State Machines," *Journal of the ACM*, **30**(2), pp. 323–342, 1983.

[3] D. Cohen & N. Dorn, "An Experiment in Analysing Switch Recovery Procedures," in *Formal Description Techniques*, **V**, ed. M. Diaz & R. Groz, North-Holland, pp. 23–34, 1993.

[4] G. J. Holzmann, *The Design and Validation of Computer Protocols*, Prentice-Hall International, 1991.

[5] H. W. Thimbleby, "An Organisational Solution to Computer Viruses," *Journal of Systems and Software*, in press, 1994.

[6] J. M. Voas, J. E. Payne & F. B. Cohen, "A model for detecting the existence of software corruption in real time," *Computers & Security*, **12**(3), pp. 275–283, 1993.