Introduction to

"Building Reliable Secure Computing Systems out of Unreliable Insecure Components"

John Dobson and Brian Randell

Department of Computing Science
University of Newcastle upon Tyne
Newcastle NE1 7RU, U.K.

During the early 1980s, much of the work on dependability and distributed systems at the University of Newcastle upon Tyne centred on the scheme we had come up with for constructing a powerful distributed system from a set of UNIX systems, taking advantage of the hierarchical naming structure used in UNIX for naming directories, files, devices, etc. We developed a means of constructing a distributed UNIX-like system out of a set of conventional centralized UNIX systems, merely by the insertion of a transparent layer of software (or what would now be called "middleware") at the system call level, so that neither the UNIX system, not any of the application programs had to be altered – this layer of software we called the "Newcastle Connection", the distributed systems we could build using it "UNIX United" systems [1]. An important characteristic of the Newcastle Connection, the chief designer of which was our colleague Lindsay Marshall, was that it dealt with *all* of the system calls, not just those involved with files, so that UNIX United was truly a distributed computing system, not merely a distributed file system.

The Newcastle team rapidly realized that it would be possible to exploit the characteristics of the Newcastle Connection and UNIX United by the provision of other transparent layers of software, including ones for load-balancing, and for hardware fault tolerance. The latter, for example, implemented a triple modular redundancy scheme, by means of which application programs could "secretly" executed in triplicate and their results voted upon. This TMR layer, which consisted of only about 700 lines of C, when run on a conventional UNIX system caused such triplicated execution to be performed in interleaved fashion on a single machine. However, if run on top of the Newcastle Connection in each of the machines in a UNIX United system, could ensure that the triplicated executions proceeded concurrently[2].

We realized that we had come up with what was essentially a recursive approach to system building[3]. This led the second author and John Rushby to the realization that such a recursion could be, so to speak, applied either to construct, or deconstruct a system. The latter we realized would enable us to construct a system that enforced a multi-level security property by allocating different security domains to different physical machines, and enforcing security constraints on inter-machine communication, rather than by means of the operating system in a single machine. In the space of a few days we had a working demonstration of our DSS (Distributed Secure System) scheme, albeit an extremely crude one in which the encryption-based security controls were implemented in software (indeed in shell-script!) rather than in small trusted special-purpose hardware communications devices - TNIUs (Trusted Network Interface Units). Moreover, we realised that given their complete independence, it would be possible to combine the use of TNIUs and TMR layers, and so produce a system that provided both multi-level security and high reliability, without having to concern ourselves with possible interference between the two mechanisms.

This work was reported to RSRE (the Royal Signals and Radar Establishment), a Ministry of Defence laboratory which was funding the contract on which John Rushby was employed at Newcastle. After some initial and very understandable skepticism, and the development of a further somewhat more realistic demonstration, RSRE decided that a full-scale prototype should be developed at their laboratory. This work was classified and for some years we were not aware of how it was progressing. However by about 1985 the RSRE DSS project had been completed and partially declassified, and we had been brought back into the arena. We then belatedly realized that, though the DSS scheme directly exploited one of our system design concepts, that relating to the use of recursion, it had completely ignored an earlier design concepts that had been developed at Newcastle, that of "ideal fault-tolerant computing components"[4].

Such components were in fact a generalization, or rather a way of explaining, the approach we had developed to exception handling, based on the extensions we had developed to our initial recovery block scheme[5]. The idea was in fact very simple: each system component should, in general, contain means of attempting to handle any exceptions that were reported to it by any components that it was using, and of reporting exceptions to the component on behalf of which it was working. Furthermore such reports should distinguish between exceptions that were due to inappropriate requests made on it, from those that arose either inside the component or from its inability to handle exceptions that were reported to it.

We found ourselves in the embarrassing position of having to report back to RSRE that, almost as soon as they had revealed their DSS system, we had realized what was to us at least, a fundamental conceptual flaw in the basic design on which it had been based. The manifestations of this flaw were that there were no clean provisions in the design for dealing with failures either of the TNIUs, or for that matter of the TMR layer. (The latter is the more obvious possibility – what should be done if the TMR voter finds that no majority exists?)

In fact the original Newcastle Connection layer had, in order to preserve transparency, needed to deal with exceptions – but could only report them in terms of the set of exceptions that were already defined by the UNIX system call interface, and which UNIX application programs already had responsibility for dealing with, however inadequately. Thus though in a UNIX United system there were additional possible causes of exceptions (e.g. related to network unreliability) such exceptions were mapped into already existing UNIX exceptions – a scheme which in fact worked pretty well, although we would not necessarily recommend it for a thoroughly engineered system.

It was while we pondered these issues, and belatedly recalled that our work was on *dependability*, and that reliability and security were merely two separate facets of this general concept, that we developed the paper that follows (which is reprinted without change, followed by a Postscript written in July 2001).

References

[1] D.R. Brownbridge, L.F. Marshall and B. Randell, "The Newcastle Connection, or - UNIXes of the World Unite!," Software Practice and Experience, vol. 12, no. 12, pp.1147-1162, 1982.

[2] L.Y. Lu. "A Virtual TMR Node," in Proc. 15th Int. Symp. on Fault-Tolerant Computing (FTCS-15), pp. 286-292, Ann Arbor, Michigan, IEEE Computer Society Press, 1985.

[3] B. Randell. "Recursively Structured Distributed Computer Systems" in Proc. 3rd Symp. on Reliability on Distributed Software and Database Systems, pp. 3-11, Clearwater Beach, Florida, IEEE, 1983.

[4] T. Anderson and P.A. Lee. Fault Tolerance: Principles and practice, Prentice Hall, 1981.

[5] P.M. Melliar-Smith and B. Randell. "Software Reliability: The role of programmed exception handling," in Proc. Conf. on Language Design For Reliable Software (ACM SIGPLAN Notices, vol. 12, no. 3, March 1977), pp. 95-100, Raleigh, ACM, 1977.

BUILDING RELIABLE SECURE COMPUTING SYSTEMS OUT OF UNRELIABLE INSECURE COMPONENTS

J. E. Dobson and B. Randell

ABSTRACT

Parallels are drawn between the problems and techniques associated with achieving high reliability, and those associated with the provision of security, in distributed computing systems. Some limitations of the concept of a Trusted Computing Base are discussed, and an alternative approach to the design of highly secure computing systems is put forward, based on fault tolerance concepts and techniques.

1. INTRODUCTION

In this paper we briefly examine links between the two apparently distinct topics of security and reliability, and argue, as has been implied by Laprie[l], that they can usefully be regarded merely as different aspects of a common problem, and so susceptible, at least in part, to common solutions. In so doing, we challenge some of the current approaches to the design of highly secure computing systems, and suggest instead that an approach which has fault tolerance as its basis can achieve a high degree of security as well as reliability.

An earlier paper by the second author[2] argued that a correct approach to systems design involves treating various issues, including reliability and security, as logically separate problems, provisions for which should be made by logically separate mechanisms. It is the thesis of this paper that, at least as concepts, reliability and security are not necessarily best treated so separately, and that their joint consideration can lead to some interesting new insights.

Some links between the subject of reliability and that of security are of course obvious, and well-acknowledged. Most notably, one would expect that due effort is spent ensuring adequate reliability of the security-critical mechanisms in any system which purported to meet any significant security criteria. Also, one would surely normally expect information which is extremely secret also to be extremely valuable, and so require it to be held reliably, as well as securely. However, we believe that the two topics have much more in common than merely being linked as requirements, and the remainder of this paper will explore these more interesting similarities.

2. DISTRIBUTED SYSTEMS

The approach to the design of highly secure systems that we are suggesting is very deliberately taking as its starting point an environment containing so-called "distributed computing systems". By this term we mean a system made up of multiple computers, interacting via a non-instantaneous communications medium, capable of working (and failing) independently of each other. Such computing systems might be spread over a large geographical area, or, in the not-too-distant future, might coexist on a single silicon wafer or even chip. But the problems of security and reliability have a wider domain than mere computers, and our real interest is in "distributed systems", i.e. in systems which could, for example, include people and machines that are interacting with, and perhaps through, one or more computing subsystems. Against such realities the problems of an isolated computing system look very narrow and uninteresting, and even treatment of distributed computing subsystems, without considering the human environment in which they are placed, can sometimes engender a feeling that what is being addressed is only a part of the real problem. Nevertheless, it is convenient to start by considering just the distributed computing component of a system.

The relevance of distributed computing systems to reliability is well-known. Though they pose additional reliability problems, they also provide the basis for a variety of reliability mechanisms. The essence of these mechanisms is redundancy and separation — the provision of additional separate means of storing, communicating and/or processing information, so that errors can be detected and recovered from, or masked, so that failures can be averted. However, as Rushby[8] and others have argued, "separation" is also at the heart of the

security problem — and manifest physical separation is one of the simplest and most easily verifiable forms of separation.

It is possible to try to build complex software and hardware mechanisms which will guarantee that a given single computer system prevents, say, top secret information leaking into unclassified files. We believe that an equivalent effect can be more easily achieved by assigning the separate component computer systems of a distributed computing system to separate security regimes, in the knowledge that the only security-critical mechanisms will then be those relatively simple and isolatable ones that are involved with inter-computer communication.

This is what has been done at the Royal Signals and Radar Establishment in the U.K., where a prototype distributed multi-level secure system has recently been constructed[4, 5]. The system uses UNIX United, and is based on the design which was originally outlined in [6]. Oversimplifying somewhat, each separate UNIX system looks like a directory in the overall system, and runs at a single security level. The rules concerning permissible information flows between levels are enforced by placing the trusted mechanisms for enforcing security policy in specially designed network interface units, which are also responsible for appropriately encrypting all inter-machine information communication. As a result, no trust has to be placed in

any of the individual UNIX systems[7]. Moreover, the overall multi-level secure system still appears to its users to be a conventional UNIX system. Thus on practical, as well as philosophical, grounds we regard security as a problem which is best addressed within the framework of distributed systems. (In fact Laprie and 'his colleagues have also recently been exploring the twin issues of reliability and security in the context of distributed computing systems. Their SATURNE project uses process replication and voting to mask processor faults, and a scheme of scattering file fragments across different storage devices in order to impede security penetrators[8].)

3. RELIABILITY

During recent years, research groups at Newcastle and elsewhere have spent much time seeking improved definitions for the various fundamental reliability concepts. Our own dissatisfaction with the hitherto typical definitions of terms such as "failure", "error" and "fault" arose early on, when we started to consider the possibility of providing what is now termed "design fault tolerance", particularly for software. We found it inappropriate to start from an enumerated list of possible faults, which had been the hardware engineers' approach — this makes little sense when one wishes to allow for the possibility that design faults, of unknown form, are still lurking somewhere in the deeper recesses of the system. Instead we took the notion of a "failure", i.e. the event of a system deviating from its specified behaviour, as our starting point, and then defined "reliability", "fault" and "error" in terms of "failure" [9].

A brief recapitulation is in order. The occurrence of a failure must be due to the presence of defects within the system. Such defects will be referred to as faults when they are internal to a component or the design and errors when the state of the whole system is defective. Even though the external state of a component may be an error in the system of which it is a part, the component need not be in an erroneous state when it is considered as a system in its own right. The internal state of the component may be perfectly valid but incompatible with the states of other components of the system. Thus the only difference between a fault and an error is with respect to the structure of the system, and the distinction between error and failure is that between a condition (or state) and an event. We can then say that a fault is the cause of an error (i.e. an erroneous state) and an error is the cause of a failure (i.e. the event of not producing behaviour as specified).

With such an approach, if one has more than one specification for a single system, each capturing some different requirement which is to be placed on its operation, one has thereby defined different types of failure, and indeed different types of reliability. However, to avoid unnecessary confusion, we will follow the lead given by Laprie and use the term "dependability" to encompass all of the different types of "reliability". This will enable us to adhere to the more common informal usage of reliability as relating to functionality and the continuity of its achievement.

Security requirements similarly result in a (partial) specification of the behaviour of a system, so that in these terms one can again regard security as a special case of dependability. This is in essence the viewpoint we will adopt in this paper. Although this may seem like playing with words, we have found the viewpoint a very helpful one, in that it has led to us to an initial, yet interesting, exploration of some of the parallels between the reliability and security areas.

The principal techniques for trying to achieve reliability can be classified into fault prevention and fault tolerance — often regarded as rival, but more profitably as complementary, techniques. Fault prevention attempts to ensure that an operational system is fault-free, either because any faults it contained were removed before it

was put it into service, or by avoiding the inclusion of faults from the outset. In contrast, the fault tolerance approach accepts the possibility that, despite whatever fault prevention efforts have been made, faults might nevertheless still be present in the operational system. In particular, fault tolerance is a way of handling residual design faults, and fault tolerance techniques such as design diversity are now becoming common in safety-critical computer applications, though more frequently for purposes of error detection than fault masking. The straightforward approach to this situation is to construct system components using masking redundancy (e.g. N-Modular Redundancy for operational faults in hardware, and N-Version Programming[10] for design faults in software) and to assume that all faults are indeed successfully masked.

A further degree of sophistication involves allowing for the possibility that such fault masking may not always be successful (or even appropriate in some cases), and equally that a given component might in any case sometimes be invoked incorrectly. Such considerations have led to the development of exception handling schemes which have the property that normal and abnormal states are clearly defined and differentiated, that the error detection and recovery mechanisms are well-structured, that exception interfaces are as well-defined as the normal ones, and that the overall scheme can be recursively applied[11]. Such schemes are therefore of potential applicability at all levels of system design. but they do depend crucially on the application of a uniform exception handling model throughout the whole system and are much more dependent on a coherent error recovery strategy than they are on the sophistication of the error detection mechanisms.

4. SECURITY

It is very noticeable to us that work on secure computing systems has typically concentrated on the use of (security) fault prevention techniques. For example, penetration exercises have been used as the basis of a fault removal strategy. However the preferred approach is to try to avoid faults, by insisting on formal or semi-formal analysis of the specification at each level, and verification that it is met by the design and its implementation. But this concentration on fault prevention, and in particular on fault avoidance, seems to apply only to the computer component of a secure environment. In a wider environment, one would expect to find plans and protocols for recognising and dealing with security leakages. After all. capturing and turning spies, for example, is in effect a form of fault location and error compensation, i.e. of fault tolerance in practice. It is perhaps surprising that more attention has not been paid to fault tolerance techniques in order to achieve security in computer-based systems.

4.1. The Trusted Computing Base Concept

The apparent assumption, at least in many research papers dealing with computer security, is that it is possible to construct totally secure computing systems, and in particular, that the security-critical software within such systems will be fault-free. Much stress has therefore been laid on the desirability of specifying and constructing a socalled "trusted computing base". One published expression of the requirements to be met by such a component is as follows:

"The trusted computing base [TCB] serves to encapsulate all the security-relevant features of a system: nothing outside the TCB can impact the security of the system, and only the TCB has to be verified"[12].

Even without questioning the practicality of achieving an absolutely secure TCB of any significant sophistication, and of doing so just by means of fault prevention techniques, there are a number of criticisms that can be made of this requirement. Firstly, a system that has to maintain security may well thereby be prevented from having the appropriate functionality for a component at that level; secondly, this choice of placement of security mechanisms may lead to the incorporation at the low levels of functions that have little value compared with the cost of placing them at those levels; and thirdly, a secure TCB might deceive a naive user into believing that the whole system is secure. We shall expand on each of these three points.

4.2. TCB Functionality

The first point is that the need for security may restrict the functionality of a Component in undesirable, as well as desirable, ways. For example, one can conceive of a security-critical I/O device whose driver, in the interests of that security, keeps no permanent records of its operation and thereby makes the task of the maintenance engineer more difficult through the lack of diagnostic information. Again, if a highly trusted user has access to a powerful and complex (and therefore, in all probability, unreliable) operating system, he will have to call on the services of at least as trusted a systems programmer when he accidentally locks up or crashes the system through the inadvertent triggering of some residual system bug. In practice, the highly privileged user should not be allowed to run the risks attendant on the use of such a complex

system, except when a suitable systems programmer is actually on hand.

Now we are not saying that these are new or unrecognised points. Our argument is that the fault tolerance approach to system dependability is intended to deal with just this kind of difficulty, where dependability through fault prevention is unrealistic because the necessary high degree of reliability can be achieved only at the cost of over-simplification. In the specific instances mentioned, one might suggest that the security critical device, if indeed it cannot be permitted to retain diagnostic information, be replicated so that a failed component could simply be discarded and replaced without affecting its overall operation; and in the second instance, one would surely have some kind of panic button which was Independent of the domain of the operating system and which when activated would initiate an automatic purge of all or part of the system. These standard mechanisms are of course nothing more than classic fault tolerance in practice. In neither case, however, is any degree of trust placed in the reliable operation of the centrally placed and more complex part of the system, only in the functioning of the simpler error detection and exception handling mechanisms — and in really critical cases, redundancy and design diversity would be applied even to these.

4.3. The End-to-End Argument

The second point Is that the desire to place security-critical functions at the lower levels of the system can lead to the incorporation at the lower levels of functions that are redundant or of little value compared with the cost of placing them there. An alternative approach is based on the observation that assignment of functions to levels is necessarily dependent on knowledge of the end applications. Since these applications are in any case going to have to perform certain functions related to security and reliability at their level, the provision of sophisticated lower level mechanisms is justified only when they enhance other features, such as performance. In the field of network communications reliability, the position has been well summarized by Wilkes:

"In computer communications there is the concept of end-to-end error control. When the importance for the computer field of data transmission first began to be realized, there was the tendency to assume that, unless telephone engineers could reduce the error rate on telephone lines to something very small, such lines would be useless for data transmission. It was soon realized that the use of redundancy with or without retransmission could make even a noisy circuit perform in an entirely satisfactory manner. Later came the insight that end-to-end error control was all that was necessary to ensure accuracy, and that reduction of the error rate across intermediate links was only useful to the extent that it improved throughput or assisted maintenance."[13]

An example of this technique which may be as appropriate to security as it is to reliability is the idea of alternate packet routing in a packet-switched network. With end-to-end error control, different packets can be sent by different routes in the interests of network load balancing. It is interesting to observe that this technique can also be used to defeat certain kinds of traffic analysis.

4.4. Security Boundaries

A third criticism that can be made of the TCB model of security encapsulation is that it may not actually correspond to what the user or application believes to be the security boundary of the system. To make this clearer, we begin with what is intended to be a diagram of a secure computer system (the diagram and argument style are taken from [14] though we are using them for different purposes).

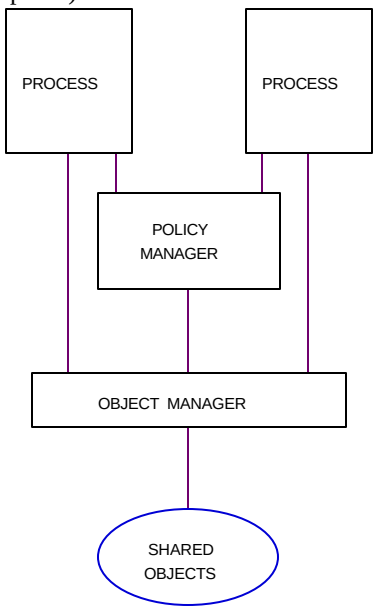

Figure 1. Typical Secure System

In Figure 1, there are a number of processes that provide various user services such as mail, editors, compilers and so on. These processes will request access to shared objects, this access being governed by an object manager according to the (abstract) security level of the process or user. The rules concerning permitted access are held by a policy manager, which also handles requests

from the user processes to change their security level. The policy manager and the object manager together can be implemented by what is referred to by the phrase 'security kernel'.

We can use this diagram to draw the security boundaries of various components described in different models, and so begin to categorize them. Figure 2 shows the security kernel interface. Figure 3 shows the TCB Interface, noting that the security architecture of a number of systems also includes "trusted processes" for certain security-critical tasks. (We note in passing that the very need for trusted processes is a justification of an earlier point, that concentration on the security features of a security kernel results in its having insufficient functionality for Its position as seen from the viewpoint of the whole secure system of which it is but a part.)

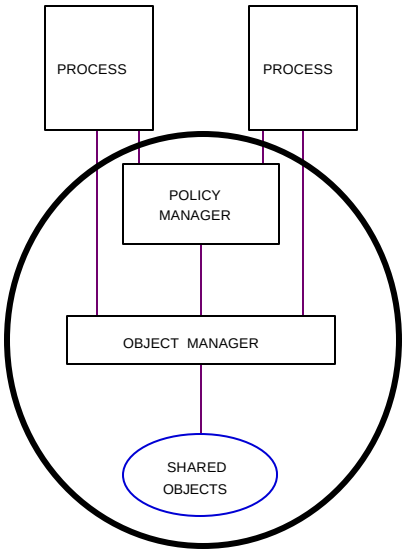

Figure 2. Security Kernel Interface

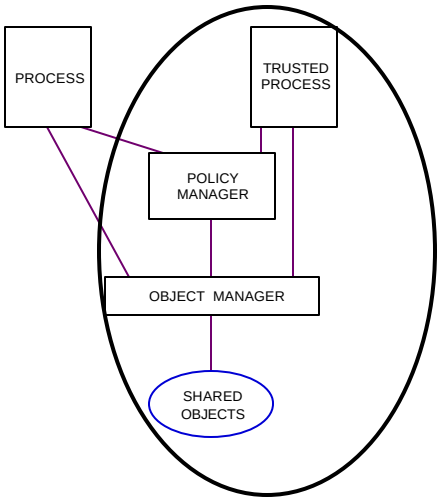

Figure 3. TCB Interface

Access control models. which are often thought to be what security is about, are indicated in Figure 4. On the other hand, the user's perception of the system Is at the interface to a program that supports a service (such as a DBMS). This program is run within the user process; consequently the security boundary divides the user process, as suggested in Figure 5.

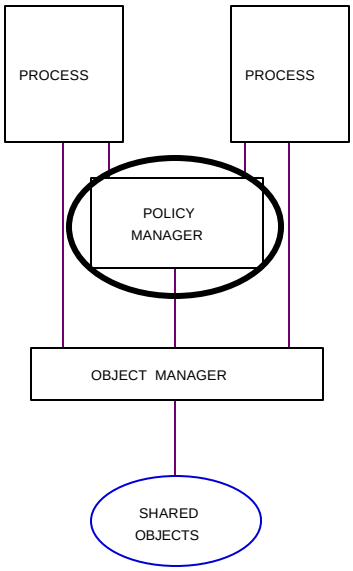

Figure 4. Security Policy interface

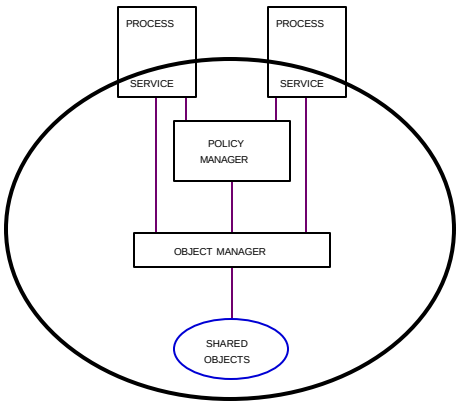

Figure 5. Application Interface

What these diagrams illustrate is the differences of interpretation of the adjective 'secure' and its derivatives, as in 'security kernel', 'security model', 'security policy', secure computing base', and so on. All the interfaces are inside, or different from, the TCB. In other words, the TCB does not always describe the user-perceived view of the security-critical behaviour of the system.

4.5. Distributed Secure Systems

There is, however, a more significant point to be made concerning the view that it is both desirable and possible to place all the security-relevant features at a single locus. Leaving aside the well-known problems associated with protection of the reliability of a single point of failure, it seems at variance with the whole notion of distributed computing systems to have such a centralised concept as an encapsulation of function. There is an alternative view of the TCB which is more relaxed, in that it regards the TCB as simply that which upholds the security properties of the system, and hence the TCB need not be represented by the kind of single closed curve interface that we have used. There are two points to be made concerning this view. The first is that our first two criticisms still hold. The second is that it will still be up to the designer to prove that the security boundary perceived from the user viewpoint does in fact correspond to the boundary perceived from the collective viewpoint of those mechanisms which together comprise the TCB. We are prepared to admit that this proof of viewpoint equivalence may be easier with a looser view of the TCB, but we are not convinced that it will be either always easy or always performed.

In any case, a centralised 'heavyweight' TCB Is not necessary for the design of a distributed secure system. What has been learnt from the design of such systems is that the separation required for security reasons can be achieved quite naturally through a combination of the distribution mechanisms and carefully placed small specialised reference monitors[6]. Such monitors will require the definition and specification of what it means to deal with security issues, und hence require also the notions of security failure, fault, error, and exception interface. A further point concerning such monitors is that in order to achieve the high degree of dependability that is required, they could be built using security fault tolerance as well as fault prevention techniques. Thus not only are the reference monitors required sufficiently simple to be rigidly specified and constructed on the base of a verified security kernel, they can also be subject to the fault tolerance techniques of design diversity and replication without the imposition of undue additional cost, and we would argue that this will in any case be required from the point of view of the most elementary dependability considerations.

Since these considerations dictate that the reference monitors themselves will require fault tolerance (if only to mask raw hardware faults), they will have to be constructed according to some architectural principle for the handling of errors and signalling of exceptions. That the importance of exceptions is not always appreciated is shown by the fact that one of the most sophisticated designs for a capability-based machine architecture, namely the Flex system[15], can permit certain exceptions to bypass the capability protections.

In fact, as mentioned earlier, where one is required to have an architectural model of dependability, this model should inform and guide the design of the whole system and not just individual components. Thus we would argue that the whole system should be built on the basis of a recursive reliability and security failure model with well-defined exception interfaces for the signalling of faults/failures and that all components, Including and especially those components on whose reliability the security of the whole system is predicated, should conform to this model. Furthermore, whatever model is employed should be capable of starting from the premise that the whole system is going to be distributed, and that not all of it is computer-based in nature. Moreover, the model should be equally applicable at all levels of abstraction and construction.

The fact that the model can be applied recursively allows its extension to an unbounded distributed system. In contrast, the model of a closed boundary within which everything has some attribute, such as security, and outside which nothing need have that attribute is, of all models, the one least susceptible to recursive definition and extension and therefore least likely to be useful as the basis of a distributed system.

It should be noted that the comments that we have been making apply not only to the construction of a

would-be secure computing system out of a set of (hardware and software) components, but also to the whole system, viewed as a component in some larger environment. Thus it is always prudent to doubt that the system is in fact as secure as its designers and certifiers allege, and also to allow for the possibility that accidental or deliberate actions by users might cause system behaviour that though "correct" with respect to the security specification, is nevertheless later found out to have been inappropriate. When these comments are taken into account, it can be seen that the secure system designer has to consider what can be done entirely within the computing system, and what aids can be given to the people in the environment of the system to assist in the forward and/or backward error recovery procedures necessary to limit damage following a security violation. (The problem is similar to that of assisting database system users who find out that their database has for some time contained, and has been giving them, incorrect data[16].)

4.6. Security Fault Tolerance

In summary, we are recommending that security be viewed just as a special case of dependability, and be the subject of similar, if not Identical, design approaches and techniques to those used for achieving high reliability. Thus we are saying that one should attempt to employ a methodology based on building secure systems out of insecure components, or more accurately, less insecure systems out of more insecure components. One should not rely totally on security fault prevention techniques, but rather one should use an appropriate blend of security fault prevention and security fault tolerance.

Our work on fault tolerance for achieving system reliability has led us to understand that the definition of reliability solely in terms of internal behaviour of a component is inadequate. As we explained earlier, one has to consider overall system state as well as the state of a particular component, and thus specify the behaviour of the system as well as of each part of it. Such specifications will characterise the desirable behaviour of the component at its interfaces with its environment, and also define any means It is designed to have of signalling such exceptions as are admitted might occur. (In principle any such specification should aim to be as complete as possible — in practice one might choose to abbreviate the specification by making it explicitly dependent on the security specifications of the sub-components and by assuming that these are not violated[17].) Every effort should be made to devise and incorporate into the system cost-effective run-time checks against possible failures to meet these specifications, as well as provisions for responding to indications that externally applied error checks have revealed security violations. Such internal and external checks should supplement any replication and majority voting schemes which are used In the system. Moreover they should be fitted into a carefully thought out, and fully general, exception handling scheme.

As the historv of the use of fault tolerance for reliability purposes has shown, experience and experiment are needed before such ideas are fully accepted, and we must admit that at this time we have no practical experiences to report on which might buttress the logical and philosophical arguments which we have attempted to marshal in this paper. The effectiveness of the approach that we are proposing would of course ideally be assessed by a controlled series of experiments, involving the implementation of systems with and without the use of such security fault tolerance techniques. Although few system structuring concepts have been made the subject of such controlled experiments, the ideas of fault tolerance have in fact by now been proven in experiment and practice, and our belief in their applicability to security and our expectation that any experiments will turn out to be worthwhile are, we would argue, soundly based in the knowledge of that history.

In support of our argument that the construction of a highly secure system is closely analogous to the construction of a highly reliable system, we now provide two examples of secure components which use reliability techniques to achieve security goals, and suggest what other reliability principles further secure systems could incorporate.

One obvious example of a practical system which provides security by using classic reliability mechanisms is a possible design of a secure file manager. Such a file system could encrypt disk blocks to prevent unauthorised access (or interpretation), mark each block with a security marking, provide a cryptographic checksum to detect tampering with the file, and physically Isolate file systems of different security levels. Here the use of encryption Is a fail-safe mechanism (even if a security failure in the form of unauthorised access does occur, no harm is done[*]), the use of security markings and checksums is just the provision of redundancy as a fault detection mechanism, and security level separation is a damage containment technique.

We would also point out that one would expect a secure file manager to have an exception signalling

[*] assuming that the encryption mechanism is itself reliable, and in particular, has not been cracked

mechanism so that faults such as a top secret disk block in an unclassified file would be dealt with at the file level rather than at the disk block level. One could then use a number of fault tolerance approaches in order to prevent a security failure (e.g. the printing of the contaminated file in an unclassified environment). For example one could reclassify the top secret level (forward error recovery) or restore the file from an archive to some previous sanitary state (backward error recovery) in addition to the fault removal task of determining why the top secret block got there In the first place.

A second example is the use of design diversity for a trusted process that Is too complex for the fault avoidance provided by full formal verification. Thus a user authenticator, for example. is an obvious candidate for N-Version Programming, since It is very clearly specifiable, has a well-defined relation between input and output, and is not subject to problems of imprecision (e.g. accuracy of floating point numbers).

These are just two examples of Individual components. The design of an entire secure system demands a more abstract level of thinking than the detailed design of the system components, and hence the reliability principles which should inform it have to be more generally expressed. We now make an initial attempt at stating a number of additional analogies which might form the basis of a secure system design.

(i) A common technique for the structuring of software to be used in reliable systems is to analyse the functionality in terms of atomic actions which either succeed or fail completely, thus permitting clean well-structured recovery mechanisms. One perhaps could use the same or analogous methods for analysing the functionality of secure systems in terms of security regions across whose boundaries guarantees can be given with respect to information flow.

(ii) A system could be designed with internal security checks (e.g. based on the use of capability mechanisms) which will enable at least some types of security violation to be prevented, and the attempt reported on, just as a fault tolerant system might contain internal error detection and exception signalling mechanisms. Such internal security checks could enable one to rely on less complete formal security proofs, by analogy with the notion of "safe" programming[l8].

(iii) A system's outputs could be, at least partly, vetted for security violations (e.g. by a real or an automated security watch officer) just as they might be subjected to acceptance tests for the purpose of determining whether to invoke error recovery.

(iv) An important task for a security watch officer is to confirm that the system has not performed completely unintended actions. This is In essence the frame problem[19]. Some recovery block implementations have provided assistance with this problem by requiring that acceptance tests must not only evaluate to true but must also access all the variables that have had assignments made to them, in order to succeed.

(v) Dependable constraints on, or mechanisms for the recording of, information flow can provide a basis for dealing with the situation when it has been determined that one or more security violations have occurred. They can be used to determine what purging mechanisms should be invoked (i.e. backward error recovery) or, at least in principle. what misinformation should be transmitted where (i.e. forward error recovery by compensation).

(vi) Certain types of security violation can also be tolerated using reliability-based mechanisms. Examples of such violations are denial of service (which is the classic unreliability problem) and signalling channels. which correspond to components that are thought by the designer to be independent but turn out not to be (and hence become suitable candidates for the application of design fault tolerance techniques).

It is perhaps appropriate to end this discussion of security, and Its close parallels to reliability, by returning to the implication, made in the Introduction, of there being an inconsistency between the present arguments and the arguments for the merits of a careful separation between security and reliability. The distinction to be made is, of course, between (i) security and reliability as characteristics or qualities, and (ii) particular security and reliability mechanisms. It is entirely natural to find that similar characteristics demand similar mechanisms. However it is also often highly advantageous to Implement distinct mechanisms (whether concerned with reliability or security) that do not need to affect a system's visible functionality, quite independently of each other.

5. CONCLUSIONS

It is not necessary to believe that security of a system requires that the system be built out of secure components, just as a reliable system is not necessarily built out of reliable components. The complexity of basic components required to achieve the functional requirements may be such as to preclude any certainty of security; alternatively, components whose security features can be adequately demonstrated may be functionally inadequate. To overcome these problems,

security fault tolerance, in addition to security fault prevention, is proposed as an approach worth investigating not only for the computer subsystem but for the whole secure system. The main features of this approach include design diversity, replication and adjudication, and above all a uniform approach to exception handling.

Nor in fact is it necessary to believe that security. as a property of a system, must follow if the system is composed of a set of components each of which is secure at the component level. This is not just for the obvious reason that insecurity might have leaked in during the construction process (security has to be defined so that it can be applied to composition as well as to a component), but for the more subtle reason that concentration on the component and construction issues might lead to the neglect of real world problems (involving both the internal computer and external human Interfaces and structures) which are visible at the level of the whole system but which it is possible to convince oneself are imaginary at the level of the detailed component.

In summary, our argument is that it should be possible to design a system which, although adequately secure at the system level, has been carefully built out of components many of which are themselves not secure. The analogy with reliability is of course that N-Modular Redundancy and other fault tolerance techniques are standard ways of building reliable systems out of unreliable components (the point being that reliable components may be unavailable or impracticable for a number of reasons). And if in fact our view that the systems problems of reliability and security have a common structure is correct, then the task of designing a system that Is adequately reliable and adequately secure requires a solution and set of mechanisms for only one abstract problem and not two.

6. ACKNOWLEDGEMENTS

Whilst developing the arguments contained in this paper, we have had considerable benefit, as well as enjoyment, from discussions with our Computing Laboratory colleagues, and Tom Anderson in particular. and also with Derek Barnes and Simon Wiseman of the Royal Signals and Radar Establishment (RSRE). The Newcastle Reliability project is sponsored by the UK Science and Engineering Research Council, and by RSRE.

References

1. J.-C. Laprie, "Dependable Computing and Fault-Tolerance," in Digest of Papers FTCS-15, pp. 2-11, June 1985.

2. B. Randell, "Recursively Structured Distributed Computing Systems," in Proc. 3rd Symp. Reliability in DistrIbuted Software and Database Systems, pp.3-11, IEEE, October 1983.

3. 3. M. Rushby, "The Design and Verification of Secure Systems, in Proc. 8th ACM Symp. on Operating Systems Principles, Asilomar, CA (ACM Operating Systems Review, Vol 15, No.5), pp. 12-20, December 1981.

4. D. H. Barnes and R. MacDonald, "A practical Distributed Secure system," in Proc. NEOS '85 (Networks and Electronic Office Systems), IERE, London, September 1985.

5. B. Wood and D. H. Barnes, "A Practical Distributed Secure System," in Proc. Int. Conf. on System Security, pp. 49-60, Online, London, October 1985.

6. J.M. Rushby and B. Randell, "A Distributed Secure System," Computer, vol.16, no.7, pp. 55-67, IEEE, July 1983. (Also: Technical Report 182, Computing Laboratory, University of Newcastle upon Tyne).

7. K. Thompson, "Reflections on Trusting Trust," Comm. ACM, vol.27, no.8, pp. 761-763, Aug.1984.

8. Y. Deswarte, J-C. Fabre, J.-C. Laprie, and D. Powell, "A Saturation Network to Tolerate Faults and Intrusions," in Proc. 5th Symp. on Reliability in Distributed Software and Database Systems, pp.74-81, IEEE, January 1986.

9. P. H. Melliar-Smith and B. Randell, "Software Reliability: The role of programmed exception handling," in Proc. of Conf. on Language Design for Reliable Software, Raleigh, NC (Sigplan Notices, Vol.12, No. 3), pp.95-100, March 1977.

10. L. Chen and A. Avizienis, "N-Version programming: A Fault-Tolerance Approach to Reliability of Software Operation," Digest of Papers FTCS-8, pp. 3-9, Toulouse. June 1978.

11. T. Anderson and P. A. Lee, Fault Tolerance: Principles and practice, Prentice-Hall, 1981.

12. K. N. Levitt, S.D. Crocker, and D. Craigen, "Introduction to VERkshop III," in Proc. VERkshop III - A Formal Verification Workshop, Pajaro Dunes, Watsonville, CA. (ACM Software Engineering Notes, Vol.10. No.4), pp. ii-iii, ACM, 18-21 February, 1985.

13. M. V. Wilkes, "Past, Present and Future of the Computer Field," lEE Proc, vol. 131 E, no.4, pp. 106-112, July 1984.

14. 3. K. Millen and C. H. Cerniglia, "Computer Security Models," Report MTR 9531, Mitre Corporation, September 1984.

15. 3. M. Foster, I. F. Currie. and P. W. Edwards, "Flex: A working computer with an architecture based on procedure values." Proc. International Workshop on High-Level Architecture, pp. 181-185, Fort Lauderdale, Florida, December 1982.

16. S. K. Shrivastava, "A Dependency, Commitment and Recovery Model for Atomic Actions," in Proc. 2nd Symp. on Reliability in Dist. Software and Database Systems, pp.112-119, IEEE, Pittsburg, July 1982.

17. R. B. Neely and J. W. Freeman, "Structuring Systems for Formal Verification," in Proc. 1985 Symp. on Security and Privacy, pp.2-13, 1985.

18. T. Anderson and R.W. Witty, "Safe programming," BIT, vol. 18, pp. 1-8, 1978.

19. E. Sandewall, "An Approach to the Frame Problem and Its ImplementatIon," in Machine Intelligence 7, ed. B. Meltzer and D. Michie. pp.195-204. Edinburgh UniversityPress, 1972.

POSTSCRIPT

The major points which are as true today as they were when we wrote the paper 15 years ago are these:

1. Security is not just an add-on property; it is an architectural property.
2. The architectural mechanisms should not be limited to defensive ones: architecture for error recovery (both forward and backward recovery) are just as important.
3. The various views of security boundaries as perceived by the various actors –users, system administrators, system security officers and so on– do not necessarily coincide and in the case of distributed systems probably necessarily do not coincide (such is the nature of distribution).

We will elaborate on each of these points in turn:

1) Our favourite example of architectural security is the medieval castle (of which there are many with 50 miles of Newcastle). Defence in depth consists of at least outer walls, moats, inner walls, keeps, and spiral staircases that put attackers (at least right-handed ones) at a severe disadvantage. Accompanying these physical structures were protocols, e.g. concerning when and where to use boiling oil, either to protect a barrier, or after it had been breached. Indeed it is these layered defences that constitute the castle as a secure system. This defence in depth is an attempt to provide reliability through diversity so as to avoid –as far as possible– a single point of failure. It is also fairly clear that the gatehouses in such castles were positioned so as to allow both ingress and egress monitoring, which can be seen as both fault prevention (keep the enemy out) and fault tolerance (if the enemy has penetrated the first defensive level, keep them from escaping).

The concept of reliability through diversity is now more important than ever in an architectural approach to security. The dangers of a monoculture are as severe in computer systems as they are in agriculture. [1]

2) A significant trend in the last 15 years has been in increasing complexity of infrastructure. This means that securing the infrastructure (for example, providing a TCB with the functionality required by applications of the complexity of today) is a much more difficult problem because the growth in functionality has been accompanied by a decrease in the understanding and use of safe programming. Elementary precautions such as checking for buffer overflow, separation of code from data, and the belief that things are benign unless proved harmful (or equivalently that software works unless it is proved not to work), coupled with the fact that wrapper techniques are not yet sufficiently advanced to deal with Web-based applications, means that error detection and recovery cannot be handled piecemeal. Exception handling has to be structured and controlled. It is not much use running through a "Have I thought of everything?" checklist after the system has been built.

This is not to say that everything has to be handled in software. People are as much a part of the system as the technical architecture, and in many cases are much better at error recovery strategies. For example, recovery from the vulnerability of the internet to distributed denial-of-service attacks could be attempted by 'fining' all the service providers whose networks were used in the flooding attacks; this would in turn force those service providers to put (possibly financial) pressure on their user system administrators to provide defensive software in the form of the most up-to-date patches [2].

3) From the security point of view, the design of the internet as an end-to-end architecture (i.e. smart terminals, dumb network) means that for many system administrators, the major system boundary is seen as lying at the TCP session or IP packet levels where many firewalls are placed, but, as pointed out by Anderson [1, p.378] this probably catches only about four of the ten most common vulnerabilities. **[Worth expanding?]** It also means that the ordinary user probably sees a boundary around the data filestore and at the perceived physical machine boundary (i.e. without distinguishing the various protocol layers that communicate across this boundary). This implies that the filestore should be checked on both writing and reading. But if it is discovered that the file being written contains a virus, or if it is discovered that a network request looks like a participation in a DDoS attack, appropriate recovery is a matter for a security policy which will cross the boundary between the technical and human domains.

In summary, then, we think that to achieve the goal today is harder than it was 15 years ago because (i) there are no systematic or disciplined solutions to protective wrapping of components of the complexity we have today; (ii) existing exception handling at the level of components is very badly practised and there is little architectural support even in architecture description

[1] Indeed, as I type this very text into my word processor, I have to print it out on an Apple laserwriter since all the HP laserwriters in the department have been rendered inoperative by the Code Red virus. That I am able to do this is a consequence of my not following local procurement policy.

[2] curiously enough, this has an analogue in social mechanisms used in early medieval villages to deal with over-grazing (an attempt to prevent "the tragedy of the commons")

languages; and (iii) it is even less widely recognised that a good security policy requires and indeed starts with good software engineering practice.

Reference

[1] R.J. Anderson, *Security Engineering*, Wiley Computer Publishing, New York, 2000