

AFRL-IF-RS-TR-2002-267
Final Technical Report
October 2002



SURVIVABILITY ARCHITECTURES

University of Virginia

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. E285

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2002-267 has been reviewed and is approved for publication.


APPROVED: WLADIMIR TIRENIN
Project Manager

FOR THE DIRECTOR:


WARREN H. DEBANY, Jr., Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 074-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE October 2002		3. REPORT TYPE AND DATES COVERED Final Oct 96 – May 00
4. TITLE AND SUBTITLE SURVIVABILITY ARCHITECTURES			5. FUNDING NUMBERS C - F30602-96-1-0314 PE - 62301E PR - E017 TA - 01 WU - 02	
6. AUTHOR(S) John C. Knight				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Department of Computer Science University of Virginia 151 Engineer's Way PO Box 400740 Charlottesville, VA 22904-4740			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/IFGB 3701 North Fairfax Drive 525 Brooks Rd Arlington, VA 22203-1714 Rome, NT 13441-4505			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2002-267	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Walt Tirenin, IFGB, 315-330-1871, tireninw@rl.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 Words) Many large information systems have evolved to a point where organizations rely heavily upon them. In some cases, such systems are so widespread and so important that the normal activities of society depend upon their continued operations. Management of transportation systems such as air traffic control, telecommunications, nationwide control of power distribution, and the financial system are examples. Military information systems are similar in that many military functions are dependent on large information systems, and the ability of the Department of Defense to use its resources effectively is contingent on the proper operation of these information systems. Such systems, both civilian and military, are referred to as critical information systems. Improving the survivability of critical information systems is essential for both civilian and military applications. The way in which this is done is to implement a monitoring and control structure known as a survivability mechanism that operates separately from the information system itself. The survivability mechanism is responsible for detecting faults and recovering from them. Important issues that arise from this approach are fast and flexible reconfiguration of the application when faults occur and protection of the survivability mechanism against security attacks.				
14. SUBJECT TERMS Critical Infrastructure Protection			15. NUMBER OF PAGES 167	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

CONTENTS

1. Introduction.....	1
2. Project Description	2
2.1. Critical Infrastructure Systems	2
2.2. Research Areas	3
3. Research Summary.....	6
3.1. Domain Analysis.....	6
3.2. Architectural Support for Network Survivability	9
3.3. Security of Survivability Mechanisms	10
3.4. Application Error Detection and Recovery.....	12
3.5. Policy Specification Languages	13
3.6. Prototyping and Demonstration System	15
4. Personnel Associated with Research Effort	18
5. Publications.....	21
6. Consulting and Advising	23
7. Discoveries, Inventions, and Patents	25
Appendix A..... “Application Error Recovery in Critical Information Systems”	26
Appendix B..... “A Security Architecture for Survivable Systems”	68
Appendix C..... “Solutions for Trust of Applications on Untrustworthy Systems”	120
Appendix D..... “A System for Experimental Research in Distributed Survivability Architectures”	147

1. INTRODUCTION

This is the final report for grant number F30602-96-1-0314, "Survivability Architectures". The period of the grant was October 1, 1996 to May 30, 2000. The principal investigator on this grant was Dr. John C. Knight of the Department of Computer Science, University of Virginia, Charlottesville, Virginia 22903.

The goal of this research was to develop techniques to improve the survivability of critical infrastructure systems. The approach taken was to use supplemental architectural structures to help cope with major events that might cause damage to the network.

To start the research process in an appropriate way, a great deal of effort was expended throughout the project to understand the applications that need to be survivable. Issues that were examined were the type of service that the application provided, the consequences of failure, and the information system architectures being used. It became clear very quickly that the applications were far more critical than we realized (and probably more than most people realize), and that the information systems were far more fragile and vulnerable than we expected.

Early ideas on the use of protection shells as an architectural concept were quickly supplanted by a more general and network-wide approach. This approach is the monitor/analyze/respond architecture that is referred to in our work as a control-system architecture. This architectural approach is very powerful in its ability to deal with events in networks that would otherwise make them fail.

The control-system architecture is also a major new security vulnerability because, were it penetrated, the adversary might gain control of the entire network. With this in mind, a major aspect of the technical work in this project was to develop ways of defending the control system itself.

In dealing with a fault, the control system associated with a critical network would, in many cases, have to make substantial changes to the application, and this leads to the notion of application error recovery. If part of the fabric of the network is lost, for example, it will be essential to make changes to the application itself perhaps including changing the services that are provided. A major part of this research project has been in determining the approaches and techniques that are required for flexible and effective application reconfiguration.

The majority of the technical results of this project are documented in papers and reports. These papers and reports are listed in section 5, and their detailed content is not duplicated here. In this report we include only a summary of the major results. We include four appendices to this report that are four technical reports which have been written to summarize some of the technical results of this project. These four reports are:

- Application Error Recovery in Critical Information Systems
- A Security Architecture for Survivable Systems
- Solutions for Trust of Applications on Untrustworthy Systems
- A System for Experimental Research in Distributed Survivability Architectures

This report is organized as follows. Section 2 provides a summary of the project in order to provide the necessary background. A summary of the research results is presented in section 3. Section 4 is a list of personnel associated with the project. Publications to date resulting from this grant are listed in section 5. Consulting and advising activities are listed in section 6, and discoveries, patents and inventions in section 7. Copies of the technical reports are included as appendices A, B, C, and D.

2. PROJECT DESCRIPTION

2.1. Critical Infrastructure Systems

Many large information systems have evolved to a point where organizations rely heavily upon them. In some cases, such systems are so widespread and so important that the normal activities of society depend upon their continued operation. Management of transportation systems such as air traffic control, telecommunications, nationwide control of power distribution, and the financial system are examples. Military information systems are similar in that many military functions are dependent on large information systems, and the ability of the Department of Defense to use its resources effectively is contingent on the proper operation of these information systems. We refer to such systems, both civilian and military, as *critical information systems*.

Societal and military dependence on these systems is growing and will continue to do so for the foreseeable future as large amounts of inexpensive computing and network hardware become available. This new hardware, particularly the communications capability that it provides, offers the opportunity to enhance existing applications in innovative ways and develop new applications. We note, however, that new applications are usually constructed in part from existing components.

Complicating the situation is the interdependence of some of these applications. For example, although limited protection against loss of power is afforded for some information systems, service following a power loss is usually severely reduced. Thus, for example, management of transportation systems will be affected significantly if there is a widespread loss of power. Similarly, loss of communication service will disrupt many other information systems such as finance, electronic commerce, and transportation. Finally, many military information systems are interconnected because basic military operations require multiple sources of information. The loss of any one can severely hamper the operations of them all.

The dependability requirements that arise with many critical information systems are quite extraordinary. For example, many current systems and others that are being planned are required to operate on networks that are distributed nationally (sometimes globally) and require twenty-four-hour-per-day, seven-day-per-week operation. In addition, these systems have to support combinations of dependability requirements. For example, they have to maintain very high levels of availability while also ensuring network-wide security.

The loss of the services that these systems provide could be very serious. Some of the consequences of failure are fairly obvious—the failure of part of the air traffic control system has manifest implications. Other consequences of failure are less obvious. The failure of certain parts of the banking system, for example, can have widespread negative impact and do so very quickly. In military applications, the loss of certain command-and-control capabilities during an action could affect the outcome severely. A number of failures of critical information systems have already been reported.

Although modern information systems provide excellent service for the most part, concerns have been raised about the possible effects of failure in critical information systems. These concerns have been heightened by the growing dependence on these systems, and by their increasing number and complexity.

Dealing with the effects of faults in information systems leads to the notion of *survivability*. Informally, by survivability we mean the ability of the system to continue to provide service (possibly degraded) when various changes occur in the operating environment. For example, when events such as hardware failure, software failure, operator error, or malicious attack occur, a critical subset of normal functionality or some

alternative functionality might be needed to mitigate the consequences of the event.

There is a need to improve the survivability of critical information systems given the increasing dependence on them, the serious consequences of their failure, and their demonstrated fragility and vulnerability. However, the approaches that can be followed to achieve this goal are limited. For example, there is little point in considering completely rewriting the software for the systems because they are just too large. Similarly, it is not possible to make drastic changes to the present system architectures. Computers and network links are performing various application functions, and this fabric is determined largely by the application itself. It is not subject to change, at least not in anything but the very long term. Finally, cost considerations dictate the use of COTS components in many circumstances even though the components might not be ideal for the application.

2.2. Research Areas

This research project has addressed the issue of increasing survivability through innovative architectural means. The emphasis was on dealing with software and security problems. The project had six major elements:

- *Domain analyses.*

The development of any new technology is only of merit if it has applicability. In order to ensure that the architectural ideas that we were pursuing were useful, we studied several typical domains (application areas). The domains were: banking and finance; air-traffic control; freight-rail control; and electric power generation and distribution. The output of these studies includes extensive details of the functionality of these applications, the computer systems upon which they operate, the vulnerabilities to which they are subject, and the system architectures that they employ.

- *Architectural support for network survivability.*

The starting point for this aspect of the project were the notions of protection shells and mediators. A shell in the sense used here is a layer of software that logically surrounds[†] a software artifact and either enforces some useful predicate on the state of the system of which the artifact is a component, supplements the functionality of the artifact in some crucial way, or acquires information about the component's actions. We refer to these actions as *policies*. The artifacts that would be surrounded in the distributed systems of interest here would be the software running on the nodes, i.e., the software on each node would be surrounded by its own shell structure.

In part, the role of a shell can be thought of as protecting an application element from a dangerous world (e.g., protection of an application from a security threat) and protecting a vulnerable world from a dangerous application (e.g., protection of the remainder of a system and its context from an application element that has been penetrated by an attacker).

One of the most important goals of this project was to extend the utility of shells to provide system-wide survivability properties. For example, individual shells can be used to ensure that individual nodes possess certain useful properties but events such as coordinated hostile attacks in which several nodes are attacked at roughly the same time are not dealt with by individual shells.

The approach that we pursued was to supplement an information system with a system to monitor and react to system-wide threats. We refer to the supplement as a *control system*. This is a significant architectural step because it moves the architectural discussion from local, i.e., a shell around a node, to at least regional and perhaps system-wide collections of cooperating shells that communicate among

[†]. Hence the origin of the term shell.

themselves entirely separately from the application communications.

- *Security of survivability mechanisms.*

The introduction of cooperating shells with hierarchic control, i.e., a survivability architecture, introduces a major new vulnerability into a critical information system. The survivability mechanism itself might be attacked, and if this attack were successful the adversary would have extensive control access to the network. Thus, security of the survivability mechanism itself is crucial.

Two major approaches to securing the survivability mechanism have been investigated. In the first, a compiler attempts to translate a source program into a binary program via a one-way function. This is referred to as one-way translation and the goal is to allow programs to be deployed that could not be reverse engineered without unreasonable expenditure of resources. Without reverse engineering, an adversary would have no way to determine what a program was doing and so would be unable to tamper with it. If successful, this approach would permit the use of trusted software on untrustworthy hosts.

The second approach to securing the survivability mechanism is to avoid all but the smallest part of the target (untrustworthy) platform and rely solely on trusted hardware. In other words, the survivability software of interest would operate on a trusted host and access information from the untrustworthy host via a direct hardware connection. This hardware is then supplemented with special-purpose hardware that provides data and instruction address checks which ensure that no tampering with trusted software occurs.

- *Application error detection and recovery.*

In the event that a major trauma occurs and an infrastructure system is affected to a point where simple techniques such as hardware replication are insufficient, the entire application software system might have to be changed to have any hope of providing service on the damaged platform. It might be necessary, for example, to systematically stop some applications, modify others, and start yet others. The capability of doing this is not inherent in even the most modern critical infrastructure systems.

To deal with the problem, it is necessary that applications be capable of recovering by reconfiguration. Precisely how applications might be organized so as to allow this was a major research topic within this project.

- *Policy specification languages.*

A major research thrust related to application error recovery was in the area of specifying that recovery. The necessary recovery actions are so complex that they cannot be specified informally with any degree of success. Thus, a unique element of the survivability architecture approach is the use of formal specification for the policies that the architecture is required to enforce, and the synthesis of the implementation from this specification. The intent is to provide: (a) a concise means of policy specification that can be analyzed using formal techniques; and (b) a means to generate shell implementations that are more easily verified.

In practice, it is clear that the magnitude of the implementation problem which arises with a survivability architecture is so great that human development of this software is entirely infeasible—there is just too much software required. Thus, a synthesis approach is the only feasible way to get the implementation completed and this is only possible if a formal specification is used.

We developed a preliminary notation for specifying policies and developed a prototype translator for the notation. The complexities of policy specification forced us to conclude that a second generation notation was needed and that was only partially completed.

- *Prototyping and demonstration system.*

For purposes of research evaluation and assessment, a prototyping system has been developed that permits models of large, critical information systems to be built and subjected to experimentation. This system implements a message-passing layer on top of Microsoft Windows 2000. This message passing layer is designed to deliver messages between hosts in the system model. On a single computer, the present system can support up to about 20,000 nodes in a system model. Vulnerabilities can be modeled as can performance based on a notion of virtual time.

A central part of evaluation and assessment is a synthetic test application that has the characteristics which have been found in complex information systems but none of the detailed functionality. The test application is based on the financial payment system within the banking network. The application runs on a network of 10,000 nodes in the prototyping system, and includes a simple model of Federal Reserve's Fedwire network, large money-center banks, and branch banks. A separate program is used to generate payment requests.

3. RESEARCH SUMMARY

In this section, we summarize the technical results of this research project. Note that these summaries are intended to provide only a brief overview since the detailed results are contained in the various papers and results detailed in section 5.

3.1. Domain Analyses

Four critical infrastructure domains were studied in depth. They were: the banking and financial system, the electric power generation and distribution system, the freight-rail transportation network, and the air-traffic-control system.

By examining specimen systems, we concluded that a major characteristic which distinguishes infrastructure systems from other networks is the way in which service is provided. Critical infrastructure systems provide service to their users by composing functionality from several nodes in a manner prescribed by the application. For example, several separate nodes in the banking network provide different parts of the requisite processing when a check is deposited. The nodes involved extend from the local branch bank to the Federal Reserve system and back to a different branch bank. From the customers' perspective, all that happens is that funds are moved from one account to another.

A second important distinction is that, as presently operated, critical infrastructure applications typically involve extensive transaction processing on networks that have a tree-like topology with nodes maintaining extensive local databases. To the end-user, the tremendous complexity is hidden and the system appears to be a single application. There is typically a very wide range of vulnerabilities.

Financial Systems

For the most part, the computer systems used by the banking industry, including the Federal Reserve, are not large networks of homogenous computers. The systems we examined are hard to summarize but two characteristics stand out: first, the systems are mostly performing transaction processing using a central node; and second, the needs of the application have led to highly heterogeneous systems that include older mainframes, desk-top machines, various local-area networks, dedicated links to affiliated organizations, and customized wide-area networks.

Such architectural heterogeneity produces a variety of unique survivability challenges but these challenges are not those with which a large homogeneous network is faced. Although some problems are shared—viruses, security attacks, and so on—the specifics are such that solutions are unlikely to be identical.

The threats faced by these systems, their relative severity, and the impact of failure do not appear to be exactly those which have been described in various reports and articles on the subject. Concern has been expressed about the potential for terrorism against the banking industry, for example. The premise is that a sophisticated attacker could penetrate the defenses of the banking industry and cause any number of problems. The impact on society of such events is assumed to be severe.

First, we note that this scenario, though possible, is far from the most serious problem as we see it. The majority of the serious problems derive from two sources:

- *Inadequate software engineering.*

The systems that we observed are built from legacy applications, new applications being developed

and modified, and purchased software. These systems run on commodity operating systems using off-the-shelf hardware. The fragility of this software is a serious issue. Significant loss of service and serious security violations are much more likely to result from software failures than from some fundamental deficiency in the associated theory.

- *Poor operational practices.*

We observed at least one instance of each of the following in the systems we examined: (1) relatively important data being transmitted in plain text over leased communications lines; (2) desk-top computers equipped with modems connected to local area networks with potential access to important data; (3) sanctioned use of Web browsers with mobile-code capacity; and (4) routine backing up of critical data with no procedures in place to restore the data at a remote site in the event that there was a failure.

With the breadth of these two areas of difficulty, the likelihood of loss of a critical infrastructure application is considerable in some cases. Yet these issues do not necessarily require any new technology to be developed.

The second major aspect of the threats that we saw in practice that differ from what has been documented is that the consequences of failure in many cases are not severe. Many of the deficiencies in the computing infrastructure of the banking system will undoubtedly lead to failures. But the consequences of these failures might not always be catastrophic for the following reasons:

- The infrastructure is partitioned in such a way that losses would not be total even if a serious security breach were to occur. Banks are independent organizations and loss of one bank would not cripple society. It seems unlikely that many banks would be lost coincidentally. There are important exceptions to this observation.
- Much of the computing infrastructure is “protected” by a broad collection of what amount to assertions (in the computer science sense) that derive from the accounting principles that the industry follows. These “checks and balances” permit detection of an extraordinary number of likely attacks by outsiders. In addition, banking system employees routinely perform sophisticated analysis of many parameters in real time looking for circumstances that might be attributable to an error.
- There is an extensive but rather ad hoc replication of facilities that the industry can use to protect itself. The provision of specialized long-haul network service (real-time credit-card authentication transactions, for example) is not limited to a single source, and so loss of one such network would not be catastrophic.

We undertook a study of securities trading, the futures and options markets, foreign exchange trading, and smart cards. All of the various markets that we studied and the payment system are implemented by a set of interconnected computer networks that extend world-wide. The foreign exchange system, for example, moves funds by electronic funds transfer between countries but funds are relayed as necessary within the United States using the Federal Reserve’s Fedwire system.

Many of the important financial markets operate in what amounts to a form of “real time”. Payments in some cases must be completed within a prescribed time limit. Credit card transactions must be authorized within bounded time, and the securities markets require that trades be confirmed in a given time.

Perhaps the most striking aspect of the financial system that has been revealed by our study is the extraordinary size of the system and the volume of transactions. Both at the level of transaction counts and in terms of the monetary value that is being manipulated, the numbers are very high. Millions of transactions of many different types are being completed per day and they represent actions involving billions of dol-

lars. These systems are extraordinarily complex and their interdependence is considerable. The vulnerabilities of the World's financial markets to attacks on their information systems is clear.

Freight Rail Transportation

We completed an extensive study of the literature and visited three major corporations: CSX Corporation; Norfolk Southern Corporation, and GE-Harris Railway Electronics Corporation. CSX and Norfolk Southern are service suppliers, and taken together they provide essentially all freight-rail service on the East Coast of the United States. GE-Harris Railway Electronics is developing advanced technology for the freight-rail industry that will permit far more efficient operation and improved safety in the next generation of train systems. An additional effect of the introduction of this new technology will be to increase the freight-rail systems dependence on its information systems.

Every freight car that moves in the United States must contain an electronic device that permits the identity of the freight car to be determined by track-side sensors. These sensors are connected to a central facility by a network so that all freight-car locations can be determined. Because there are thousands of freight cars in the United States and because they are moving, the tracking process involves enormous amounts of data storage and manipulation.

Along with freight car information, freight-rail information systems maintain details of the status of locomotives, crews, and payloads. Many important applications use the databases of train information. Scheduling, for example, is carried out in such a way that payloads arrive when they are needed—so-called “just-in-time” delivery. Switching is a second application using the databases although actual authority to move is communicated to trains by human operators.

The databases used in freight-rail control are maintained in data centers that the operating companies own. These data centers use a variety of techniques to enhance availability including replicated data, redundant computing facilities, back-up power, and physical protection.

As a result of this study, we have concluded that freight-rail information systems are one of the most critical information systems in the infrastructure. The loss of one of these systems for any reason could lead to widespread disruption of energy production, manufacturing, food distribution, and many other important services. It is also significant that the civilian freight-rail system is used by the Department of Defense to move many military supplies.

Electric Power Generation and Control

The equipment associated with any given utility company is connected to the equipment of every other utility company through the national electric grid. Power is moved from one region to another over transmission lines to deal with variations in demand and variations in the cost of generation. The power flowing along any particular line changes by the minute and is not under the control of the utility that owns the line.

A given utility's generators are required to operate at a frequency that, averaged over time, gives a net flow of zero into the utility's operating region. Monitoring and managing the generating plant is under the control of a computer system. In addition, provision is made to deal with loss of generating equipment so that switching is achieved by a computerized control system also. Most modern generating equipment provides a network interface that allows direct computer control of the equipment.

A national computer database facility is maintained by Ontario Hydroelectric Co (Canada) that records the

state of transmission lines and the load that the lines are carrying on a national basis. Utilities can enquire (in real time) of this database to determine what is flowing through any given transmission line and where the power is coming from and going to (i.e., who is selling power to whom).

The electric power industry is in a state of flux as a result of deregulation. Power generation and transmission are to be considered separate functions and will be marketed as separate services. This separation requires that utilities separate their operations and associated computing systems. This process is incomplete at this time.

Utility industry organizations are in the process of defining the protocols to be used in the next generation of control systems in which utility computer systems will be fully interconnected. The vulnerability of current systems to loss of service through some form of computer-system failure is limited but growing. The report that an NSA team was able to disrupt a large fraction of the United-States power grid (see, for example, the report on cyber crime by the Center for Strategic and International Studies, www.csis.org) is almost certainly an erroneous report. However, there are indications that the integrated national electric control system will be a significant source of vulnerability and the current planning being undertaken by the industry does not take survivability issues into consideration.

Military Information Systems

Just as civilian systems are critical infrastructures, so many systems that are owned and operated by the Department of Defense are critical. The command and control functions that are central to military operations are based on information systems that are basically large distributed systems.

We studied some elements of GCCS (most of GCCS is classified) and concluded that the issues that arise in such systems are essentially the same as the ones that we see in civilian systems. Thus, we conclude that: (a) techniques developed as part of this project are probably of direct relevance to the Department of Defense's systems, and (b) the civilian systems that we have studied are likely areas that will be attacked in future wars. Both military and civilian information systems need to be protected against the possibility of information warfare and this needs to be undertaken quickly.

3.2. Architectural Support for Network Survivability

A significant goal of this project was to investigate techniques for the provision of network-wide survivability properties. The primary mechanism by which survivability will be obtained is *fault tolerance*. Thus, we sought to develop what amount to fault-tolerant network applications. Faults of interest include all forms of hardware and software failure, all forms of operator and operational errors, and security penetrations—more formally known as *deliberate faults*.

When affected by a non-local fault, a critical information system must be adapted so as to continue to provide information services on which infrastructure service depends. An important issue, therefore, is how this can be done. The traditional approach to maintaining service in complex systems (such as avionics platforms) is the use of explicit control. Our approach to survivability architectures is based on the use of explicit control to manage information systems. In essence, such a control system is responsible for choosing a configuration for the critical information system at each point in time based on current conditions to minimize the loss of service, with assurances that under defined circumstances service will meet the survivability requirements.

The need for a particular form of service at any given time is determined by application experts and amounts to a system requirement. A survivability specification documents the service requirements that

need to be met with various probabilities. The design space determines the extent to which a control system can manage loss of services when faults manifest themselves. An information system for a survivable infrastructure system must have a configuration-enabling service provision that meets the various service requirements with each defined probability.

The key characteristics of the architectural style that we have developed are that it effects adaptive control and is decentralized, hierarchical, and discrete-state. A control system manipulates a controlled system on the basis of sensor data from the controlled system, predictions of its behavior, and other such information to maintain acceptable levels of system operation. A decentralized control system is one in which parts of the control system control parts of the underlying system autonomously. An adaptive control system is one that can continue providing control in the face of changes to the controlled system and to the control system. For example, an adaptive control system for an avionics application can ensure that an aircraft remains under control even if part of a wing is damaged in flight, provided the damage is within the boundaries assumed in the adaptation.

A hierarchical control system is one in which control actions are determined at a number of levels in a hierarchical system, with low-level control system elements influencing and being influenced by higher levels of control. Tactical decisions might be made close to individual components in a controlled system, while strategic decisions are made at a higher level based on aggregated wide-area system state. For our purposes, the controlled system is the information system supporting an infrastructure system. In the case of freight rail, for example, the physical system comprises rails, cars and locomotives. This infrastructure is controlled by a complex information system that manages train assembly, dispatch, scheduling, move authority, billing and so on to meet performance, safety, business and other objectives. A survivability architecture would supplement the information system with a survivability control system.

A wide variety of survivability properties can result from the non-local fault tolerance achieved by this structure including intrusion monitoring and response and controlled service degradation under adverse conditions. The need for a hierarchical structure is implied by the size and distribution of infrastructure information systems. It is implausible, for example, to have a single computing node monitoring the entire United States banking system. Each major bank would have a local control system interacting through abstract interfaces with higher-level (e.g., Federal Reserve) and lower-level (e.g., branch) control systems. A hierarchical structure is natural to support scalability through local control and the passing of aggregated status information up and down a hierarchy. Such information flows will be needed in practice to implement non-local reconfiguration policies with acceptable performance. Such a structure enables local control nodes to implement policies based on local information and aggregated global state passed from above. In addition to performance, hierarchy enables abstraction and complexity control in control system implementation. Details of local application nodes are abstracted by local control nodes. Higher-level control nodes are specified and implemented in terms of the observable and controllable aspects of control nodes at the next level down the control hierarchy. Hierarchy is also intended to foster evolvability of the control policies. Such evolvability will be critical to effective response by a system to threats that change over time and as the underlying information and infrastructure systems evolve.

3.3. Security of Survivability Mechanisms

The security of the survivability architecture is a crucial aspect of this research. If the survivability architecture is not secure, then it introduces a major vulnerability. Two separate technical directions have been pursued in order to try to deal with this problem. They are *code obfuscation* and *hardware monitoring*.

Code Obfuscation

The code obfuscation approach being developed is based on a *one-way translation* technique, i.e., a means to translate the source code for the trusted software into a binary image in such a way that the reverse transformation cannot be determined without the expenditure of tremendous resources. Thus, even if an adversary has the source code for the trusted software and can obtain a memory image, he or she will not be able to determine the locations of variables, the locations of functions, or the protocols being used. Without this information, the adversary will not be able ascertain what data is being transmitted out of the trusted software, how it is being transmitted, or when.

Although the one-way transformation reduces the probability that an adversary can manipulate the binary image of the software to a very low value, it is conceivable that an adversary could reverse engineer the binary image by what would amount to a systematic state-space exploration. This is much like a brute-force attack against encrypted information. To prevent this form of attack, we have developed mechanisms for period replacement of the binary image and periodic re-randomization of the binary image during execution.

The way in which the source-to-binary transformation will be made essentially one-way is by randomization during the translation process. We have developed techniques in which randomization is applied to an intermediate representation of the program. The final binary image will have been derived from the source program using random processes that depend on random-number seeds that function as keys to the transformation.

We have developed a translator that incorporates all the transformation concepts and with which these concepts have been evaluated. The translator is a source-to-source translator for C that produces a tree representation of the source program as its intermediate form. The translator also gathers statistics about the source-program structure so as to provide the necessary input for randomization algorithms. The SUIF2 toolkit from Stanford University that was produced as part of the National Compiler Infrastructure project was used to develop the translator.

Research on randomization algorithms has focused on the development of code transformations that introduce constructs into source programs that are known to be extremely hard to analyze. An example is aliasing. By introducing aliasing in large amounts into programs, we have been able to produce programs that are semantically equivalent to the original yet are beyond any current form of static analysis.

Hardware Monitoring

The hardware monitoring approach derives from the goal of using basic hardware memory protection to secure the trusted software. In principle, the problem we face should be addressed by the facilities found in most memory-system designs. Blocks of storage can be designated as execute-only, read-only, and so on in most modern processors. Unfortunately, the control of these properties lies with the software and so, for security purposes, they are for the most part useless.

Given this situation, we sought ways of isolating the memory protection mechanism so that it could be used. One strategy that does work, at least in part, is to use a read-only memory for the address space that contains the software which controls the actually hardware memory protection mechanism. This approach fails however, when one considers the data that this software has to use—it has to be writable by definition—and when one examines the possibility of rogue software executing instructions that affect the protection control registers. In practice, this type of approach is very complex and it is very hard to prove that such approaches are effective.

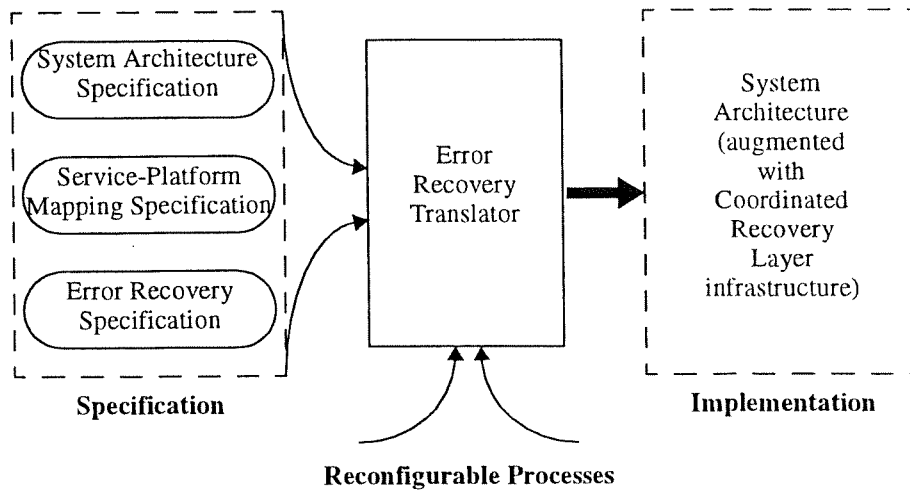


Figure 1. Error recovery methodology.

The hardware solution that we have developed takes a more radical approach and places the trusted code on a separate *trusted* host that is collocated with the *untrustworthy* host. This trusted host need not have a lot of performance and can be physically secured against tampering. It need not have any network connection. The trusted host is connected to the untrustworthy host at the level of the memory bus and its role is to perform real-time checks on the addresses being generated by the untrustworthy processor. By this means all memory references are checked against bounds in real time where the checks themselves cannot be affected by any tampering on the untrustworthy machine. The only facility required to enable this approach to work is for the memory system on the untrustworthy host to be trusted. Since we can reasonably assume that hardware tampering is much harder than software tampering, this seems like a reasonable assumption. If it is not, then basically no security approach will work.

3.4. Application Error Detection and Recovery

To make a critical application fault tolerant, it is necessary to introduce mechanisms to recognize the errors of interest, maintain state information about the system to the extent that it affects error recovery, and define the required error recovery from all possible system states. The size of current and expected critical information systems, the variety and sophistication of the services they provide, and the complexity of the reconfiguration requirements mean that an approach to fault tolerance that depends upon traditional software development techniques is infeasible in all but the simplest cases. The likelihood is that future systems will involve at least tens of thousands of nodes, have to tolerate dozens, perhaps hundreds, of different types of fault, and have to support applications that provide very elaborate user services. Programming fault tolerance for such a system using conventional methods is quite impractical.

Overview

An overview of our solution approach is shown in Figure 1. Our solution approach is based on the use of a formal specification to describe the required error detection and error recovery, and the use of synthesis to generate the implementation from the formal specification. The specification is broken down into a variety of parts that define different aspects of the problem but a unique element of the approach is the use

of a *multi-level* specification. The required error detection and error recovery specifications are written in terms of high-level abstract entities that are defined in an intermediate specification that exports high-level entities but is itself written in terms of the low-level system components.

In addition to the use of formal specification and synthesis, a system architecture supporting error recovery must be put in place. At the node level, each node must be constructed such that it supports reconfiguration and incorporates the generated code that implements reconfiguration. At the system level, coordination and control services must be provided to the reconfiguring nodes; a global entity called the Coordinated Recovery Layer provides these support services.

Finally, the use of formal specification permits various forms of analyses to check the error recovery algorithms. Various forms of syntactic and semantic analyses are enabled by the formal specification notations, including invariant assertion checking.

Issues in Specification and Synthesis of Error Recovery

The first issue of concern in formally specifying error recovery is determining the components required in a specification language. Clearly, a finite-state machine must be constructed to specify the initial configuration of the system and any reconfigurations required to recover from system errors. This finite-state machine consists of all possible states the system could be in given the system errors that are of interest and should be handled, and the activities to undertake on transition from one system state to another. Two key aspects of the system must be described: (1) the system architecture and (2) the services that the nodes of the system provide.

Even for a simple system, the error recovery specification becomes *very* large and unwieldy, and this is the key problem that had to be addressed. Three approaches can be followed to deal with the state explosion problem:

- The specification itself could be constructed in such a way as to keep it manageable: for example, portions of the system could be abstracted and consolidated into single objects in the specification, thus ensuring that the specification deals with very small numbers of objects regardless of how many actual nodes there are in the system.
- The specification notation could be enhanced to accommodate larger numbers of nodes. One way of achieving this would be to introduce and integrate some form of set-based notation to enable description and manipulation of large numbers of nodes simultaneously.
- The system description could be contained in separate databases, independent of the error recovery specification. It is likely that there already exists in some form voluminous descriptions of these critical systems; in whatever form these system descriptions exist, our error recovery system can integrate and manipulate those descriptions for the purposes of specifying error recovery.

These thoughts led to the RAPTOR specification language.

3.5. Policy Specification Languages

The first-generation RAPTOR specification notation consists of four major sub-specifications:

- *Error Detection Specification (EDS)*

The error-detection specification defines the overall systems states that are associated with the various faults of interest.

- *Error Recovery Specification (ERS)*

The error-recovery specification defines the necessary state changes from any acceptable system reconfiguration to any other in terms of topology, functionality, and geometry (assignment of services to nodes).

- *System Architecture Specification (SAS)*

The system architecture specification describes the topology of the system and platform including the computing nodes, the communications links, and detailed parametric information for key characteristics. For example, nodes are named and described additionally with node type, hardware details, operating system, software versions, and so on. Links are specified with connection type and bandwidth capabilities.

- *Service-Platform Mapping Specification (SPMS)*

The service-platform mapping specification relates the names of programs to the node names described in the SAS. The program descriptions in the SPMS include the services that each program provides, including alternate and degraded service modes.

As noted above, some form of accumulation of states or other simplification must be imposed if an approach even to specification of fault tolerance is to be tractable. The key to this simplification lies in the fact that many nodes in large networks, even those providing critical infrastructure service, do not need to be distinguished for purposes of fault tolerance. In the banking application, for example, it is clear that the loss of computing service at any single branch is both largely insignificant and largely independent of which branch is involved. Conversely, the loss of even one of the main Federal Reserve computing or communications centers would impede the financial system dramatically—some nodes are *much* more critical than others. However, the loss of 10,000 branch banks (for example because of a common-mode software error) would be extremely serious—even non-critical nodes have an impact if sufficient of them are lost at the same time.

To cope with the required accumulation of states, the overall specification is made two-level, and we add a fifth element to the specification approach. The SAS and the SPMS are declarative specifications, and in practice the content of these specifications are databases of facts about the system architecture and configuration. The EDS and ERS are both algorithmic specifications—they describe algorithms that have to be executed to perform error detection and recovery respectively. In principle, these algorithms can be written using the information contained in the SAS and SPMS. But it is precisely this possibility that leads to the state explosion in specification. The SAS and SPMS are just too big.

The fifth element is the *system interface specification (SIS)*. This is a specification that defines major system objects in terms of the lower-level entities contained in the SAS and SPMS. These objects are exported from the SAS and SPMS and become the objects with which the EDS and ERS are written. This structure is shown in Figure 2.

The overall structure of the EDS is that of a (traditional) finite-state machine that characterizes fault conditions as states (defined using sets) and the requisite responses to each fault are associated with state transitions. Arcs are labeled with faults and show the state transitions for each fault from every relevant state. The actions associated with any given transition are in the ERS and are extensive because each action is essentially a high-level program that implements the error-recovery component of the full system survivability specification. The complete system-survivability specification documents the different states (system environments) that the system can be in, including the errors that will be detected and handled.

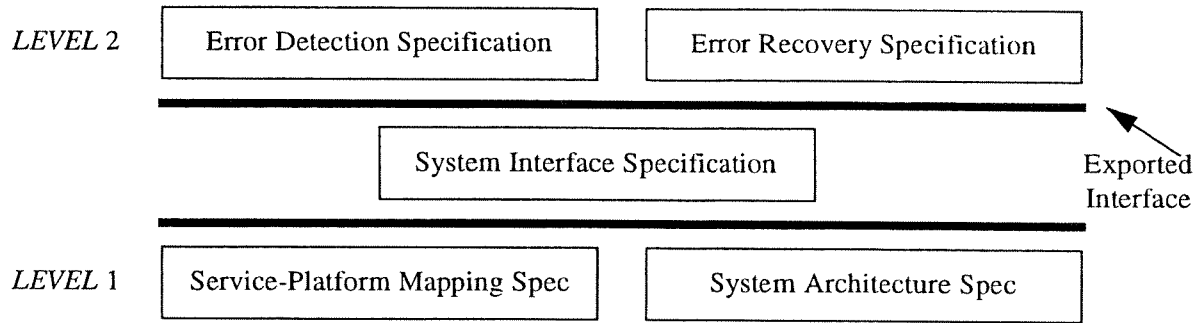


Figure 2. Two-level specification structure.

3.6. Prototyping and Demonstration System

Experimentation is a crucial element of research in this area. Unfortunately, several factors present serious impediments to experimentation. First, infrastructure systems are privately owned. Second, they are, by definition, critical from both the business and societal perspectives. It is inconceivable that their operators would permit experimentation on them. Third, infrastructures are enormous in physical scale, cost and complexity, which makes it infeasible to replicate them in the laboratory.

Given the impediments to direct experimentation with real infrastructures, we have adopted an experimental approach based on operational models. Building operational models of critical infrastructure information systems presents two significant challenges: (1) modeling the critical but no other aspects of infrastructure systems with sufficient accuracy and completeness; and (2) facilitating inclusion in the model of relevant architectural mechanisms to be developed or evaluated. For purposes of experimentation, an operational model has to represent relevant functional and architectural features of a given system, as well as its operational environment, including a dynamic model of internal failures and external threats. Once such a model is built, mechanisms must be present to allow prototypes of architectural survivability mechanisms to be introduced. Both models and architectural supplements must be instrumented for collection of data needed to analyze and evaluate survivability mechanisms.

The system we have developed to meet the various requirements outlined above is called RAPTOR. For purposes of experimentation, the RAPTOR system provides the user with an efficient, easily manipulated operational model of a distributed application with extensive control, monitoring, and display facilities. Figure 3 provides an overview of the system.

A RAPTOR model is specified by defining the desired *topology* and the desired *application functionality*. From the topology, the model is created using services from the modeling system's support libraries and using application software provided by the model builder. *Vulnerabilities* to which the model should be subject are defined and controlled by a user-defined vulnerability specification. During the execution of a model, *symptoms* can be injected into the model to indicate any event of interest to the user. Events might include security penetrations, hardware failures, etc. Any *data* of interest to the user can be collected and made available to a separate process (possibly on a remote computer) for *display* and analysis. Finally, since multiple independent models can be defined from separate topology specifications, complex systems

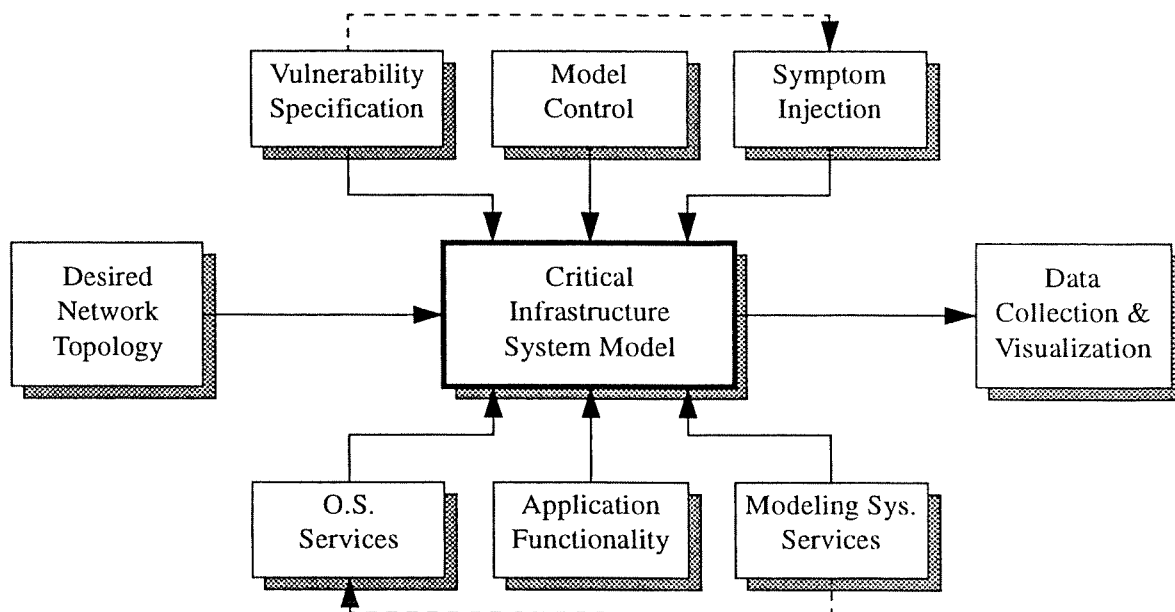


Figure 3. The RAPTOR modeling system architecture

of *interdependent* critical networks can be modeled (see Figure 4).

The basic semantics of a model is a set of concurrent message-passing entities that we refer to as *virtual message processors*. Figure 5 depicts the general structure of a virtual message processor. A virtual message processor is provided with a queue of incoming messages that it can read and process as it chooses. Usually, these messages are routed from the input queue to programmable message interpreters. Any new messages generated as a result of interpreting received messages are sent immediately although their arrival times at their destinations can be controlled. Messages can be generated asynchronously also if needed based on, for example, a timer event.

Within the modeling system, a network node is modeled as a set of one or more virtual message processors each of which is executed by a separate OS-level thread. A typical simple node can be modeled with a single virtual message processor and hence a single thread. More complex nodes can be modeled as a collection of threads thereby allowing such nodes to exhibit concurrent internal behavior. The use of threads for

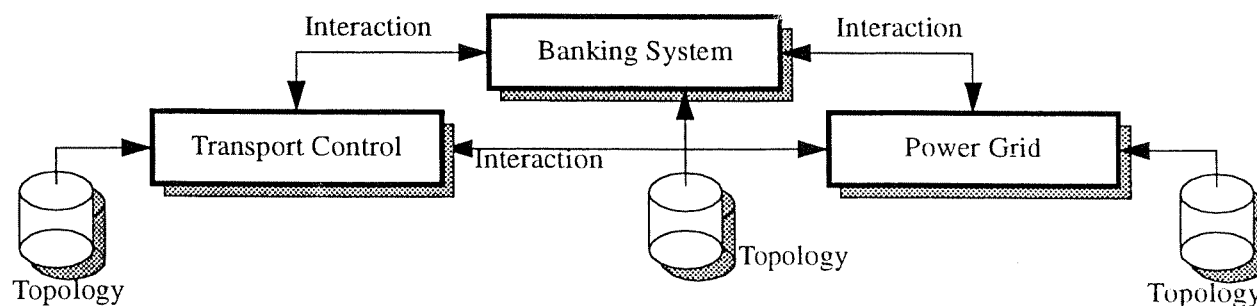


Figure 4. A Model of Multiple Interacting Infrastructure Systems

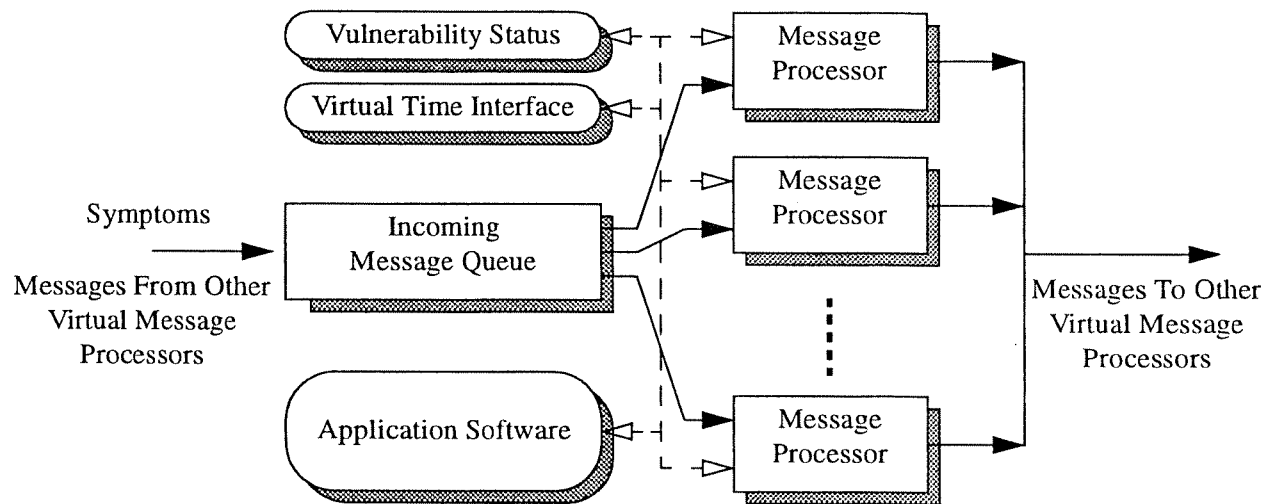


Figure 5. RAPTOR Virtual Message Processor

modeling nodes allows the model of a single node to be itself concurrent thereby permitting issues such as synchronization errors and race conditions to be modeled realistically.

Node-to-node communication is modeled by message passing between threads. Messages are passed through memory and so message passing is very efficient. Any thread can send a message to any other thread subject to restrictions imposed by a model's topology (see below). In order to permit models in which resource contention might occur, the input message queue for a virtual message processor can be given a maximum size. If the input message queue is full when a message arrives, the attempt to send the message to that virtual message processor fails. Message transmission times can also be specified to allow transmission delays to be modeled.

4. PERSONNEL ASSOCIATED WITH RESEARCH EFFORT

During the period of performance of this research effort the following faculty and professional staff were associated with the project:

Name	-	John C. Knight
Position	-	Professor
Institution	-	Dept. of Computer Science, University of Virginia

Name	-	Kevin J. Sullivan
Position	-	Associate Professor
Institution	-	Dept. of Computer Science, University of Virginia

Name	-	Robert F. Trent
Position	-	Professor (retired)
Institution	-	McIntire School of Commerce, University of Virginia

Name	-	John McHugh
Position	-	Professor
Institution	-	Dept. of Computer Science, Portland State University

Name	-	Xing Du
Position	-	Research Scientist
Institution	-	Dept. of Computer Science, University of Virginia

Name	-	Raymond W. Lubinsky
Position	-	Research Scientist
Institution	-	Dept. of Computer Science, University of Virginia

Name	-	Sandy Bryant
Position	-	Research Scientist
Institution	-	Dept. of Computer Science, University of Virginia

The following graduate students were supported in part under this grant:

Name	-	Jonathan C. Hill
Advisor	-	Jack W. Davidson
Thesis/Dissert. Title	-	To be determined
Degree	-	Ph.D.
Status	-	In progress, expected May, 2002

Name	-	Matthew Elder
Advisor	-	John C. Knight
Thesis/Dissert. Title	-	Error Recovery in Critical Infrastructure Systems
Degree	-	Ph.D.
Status	-	In progress, expected May, 2001

Name	-	Chenxi Wang
Advisor	-	John C. Knight
Thesis/Dissert. Title	-	A Security Architecture For Survivable Systems
Degree	-	Ph.D.

Status	-	In progress, expected September, 2000
Name	-	Timothy Bellaire
Advisor	-	John C. Knight
Degree	-	M.C.S.
Status	-	Graduated May, 2000
Name	-	E-Ching Lee
Advisor	-	Kevin J. Sullivan
Degree	-	M.C.S.
Status	-	Graduated May, 2000
Name	-	Luis Nakano
Advisor	-	John C. Knight
Degree	-	M.C.S
Status	-	Graduated December, 1999
Name	-	Steven Geist
Advisor	-	Kevin J. Sullivan
Thesis/Dissert. Title	-	Experimental Evaluation of an Information Survivability Control System Model
Degree	-	M.S.
Status	-	Graduated May, 1999
Name	-	Robert Sielkin
Advisor	-	Anita K. Jones
Thesis/Dissert. Title	-	Application-Specific Intrusion Detection
Degree	-	M.S.
Status	-	Graduated May, 1999
Name	-	Mark Marchukov
Advisor	-	Kevin J. Sullivan
Dissertation Title	-	N/A
Degree	-	Ph.D.
Status	-	Left the program
Name	-	James Flinn
Advisor	-	Robert F. Trent
Degree	-	M.S.I.S.
Status	-	Graduated August, 1998
Name	-	Patrick Marx
Advisor	-	Robert F. Trent
Degree	-	M.S.I.S.
Status	-	Graduated August, 1998
Name	-	William Dixon
Advisor	-	John C. Knight
Degree	-	M.C.S
Status	-	Graduated May, 1997

In addition to the graduate students listed above, several undergraduates were funded in part as research

assistants by this research project. The students were: Brian Hicks, Ryan McGeary, Paul Shaw, Sarah Schwarm, Shana Schwartz, Aaron Schwartzbard, and Kin-Way Yim.

5. PUBLICATIONS

During the period of this research project, the papers contained in the following list were prepared. Copies have been provided under separate cover:

1. Wang, C., J. Knight, "A Security Architecture for Survivable Systems", Technical Report CS-2000-26, Department of Computer Science, University of Virginia, August 2000.
2. Elder, M, J. Knight, "Application Error Recovery in Critical Information Systems", Technical Report CS-2000-28, Department of Computer Science, University of Virginia, August 2000.
3. Hill, J., J. Knight, J. Davidson, "Solutions for Trust of Applications on Untrustworthy Systems", Technical Report CS-2000-27, Department of Computer Science, University of Virginia, August 2000.
4. Knight, J., R. Schutt, K. Sullivan, "A System for Experimental Research in Distributed Survivability Architectures", Technical Report CS-2000-29, Department of Computer Science, University of Virginia, August 2000.
5. Wang, C., J. Knight, "Towards Secure Intrusion Detection", submitted to the Third Information Survivability Workshop, ISW 2000, Boston MA.
6. Knight, J., K. Sullivan, "Towards a Definition of Survivability", submitted to the Third Information Survivability Workshop, ISW 2000, Boston MA.
7. Wang, C, J. Hill, J. Knight, J. Davidson, "Protection of trusted probe programs against untrustworthy hosts", Submitted to the 2001 Network and Distributed System Security Symposium. February, 2001. San Diego. California.
8. Wang, C., J. Knight, M. Elder, "On Computer Viral Infection and the Effect of Immunization", To appear in the 16th Annual Computer Security Applications Conference, December 2000, New Orleans, Louisiana.
9. Knight, J., K. Sullivan, "Survivability Architectures: Issues and Approaches", Proceedings of DISCEX 2000, DARPA Information Survivability Conference and Exposition, IEEE Computer Society Press, Los Alamitos, CA, pp. 157-171, January 2000.
10. Sullivan, K., J. Knight, X. Du, S. Geist, "Information Survivability Control Systems", Proceedings of the International Conference on Software Engineering, IEEE Computer Society Press, pp. 184-192, May 1999.
11. Knight, J., M. Elder, and X. Du, "Error Recovery in Critical Infrastructure Systems", Proceedings of CSDA '98: Computer Security, Dependability and Assurance—From Needs to Solutions, IEEE Computer Society Press, Los Alamitos, CA, pp. 49-71, 1999.
12. Knight, J., K. Sullivan, J. McHugh, S. Geist, "A Framework for Experimental Systems Research in Distributed Survivability Architectures", Technical Report TR 98-38, Department of Computer Science, University of Virginia, December 1998.
13. Elder, M.C., J.C. Knight, "Major Security Attacks on Critical Infrastructure Systems", submitted to IEEE Computer.
14. Knight, J.C., M.C. Elder, J. Flinn, and P. Marx, "Analysis of Four Critical Infrastructure Applications", University of Virginia, Department of Computer Science, Technical Report 97-

27, (November 1997, revised September 1998).

15. Knight J.C., M.C. Elder, A.C. Chapin, B.K. Combs, Brownell, S. Geist, Steven, S. McCulloch, L.G. Nakano, R.S. Sielken, "Topics in Information Survivability" University of Virginia, Department of Computer Science, Technical Report 98-22, (August 1998).
16. Knight, J.C., R.W. Lubinsky, J. McHugh, and K.J. Sullivan, "Architectural Approaches to Information Survivability", University of Virginia, Department of Computer Science, Technical Report 97-25, (November 1997).
17. Knight, J.C., Position paper, Panel on Wrappers, Composition, and Architecture Issues for Security and Survivability, 20th National Information Systems Security Conference, Baltimore, MD, (October 1997).
18. Chalasani, P., S. Jha and K. Sullivan, "An Options Approach to Software Prototyping Decisions," Carnegie Mellon University, Department of Computer Science, Technical Report CMU-CS-97-161, (July, 1997).
19. Knight, J.C., Position paper, "Is Information Survivability an Oxymoron?", COMPASS '97: Conference on Computer Assurance, NIST Gaithersburg, MD, (June, 1997).
20. Sullivan, K.J., P. Chalasani and S. Jha, "Software Design Decisions as Real Options," University of Virginia, Department of Computer Science, Technical Report 97-14, (June 3, 1997) (submitted to IEEE Transactions on Software Engineering).
21. Sullivan, K.J., J. Socha, and M. Marchukov, "Using Formal Methods to Reason about Architectural Standards", ICSE '97—International Conference on Software Engineering, Boston MA (May 1997).
22. Sullivan, K.J. and M. Marchukov, "Interface Negotiation and Efficient Reuse: A Relaxed Theory of the Component Object Model," University of Virginia, Department of Computer Science, Technical Report 97-11 (May, 1997).
23. Knight, J.C. and K.J. Sullivan, Position paper, "Survivability Architectures", Information Survivability Workshop, SEI/CERT, San Diego, CA (February, 1997).

6. CONSULTATION AND ADVISING

As a result of this research effort, general discussions have been held with a variety of industrial and government personnel. These discussions were usually two-way. Project personnel were: (a) able to learn detailed information about survivability in specific areas; and (b) able to provide advice and comment about this project's goals and results.

Specific discussions with government or military personnel that were held or related presentations that were given include the following:

Contact	-	Mark LeBlanc
Organization	-	U.S. Department of State
Dates	-	Various
Subject matter	-	General Survivability Issues
Contact	-	Steve Rinaldi
Organization	-	U.S. Office of Science and Technology Policy
Dates	-	Various
Subject matter	-	General Survivability Issues
Contact	-	Bruce Summers
Organization	-	Federal Reserve Automation Services
Dates	-	Various
Subject matter	-	Banking Information Systems
Contact	-	William Thompson
Organization	-	Virginia Power
Dates	-	Various
Subject matter	-	Electric Power Information Systems
Contact	-	Gordon Sinkez
Organization	-	CSX Corporation
Dates	-	Various
Subject matter	-	Railway Information Systems
Contact	-	Bruce Haddan
Organization	-	Norfolk Southern Corporation
Dates	-	Various
Subject matter	-	Railway Information Systems
Contact	-	Thomas Longstaff
Organization	-	Software Engineering Institute
Dates	-	Various
Subject matter	-	Survivability Technology
Contact	-	Various
Organization	-	Information Technology for Crises Management Team of the Federal Information Services and Applications Council
Date	-	October 1999
Subject matter	-	Seminar on Critical Infrastructure Domains

Contact	-	Various
Organization	-	U.S. Department of Commerce
Date	-	September 1998
Subject matter	-	Survivability Technology
Contact	-	Various
Organization	-	Quantum Research International
Date	-	May 1998
Subject matter	-	U.S. Army Information Systems
Contact	-	Various
Organization	-	McIntire School of Commerce
Date	-	July 1997
Subject matter	-	Survivability/Security Seminar for Business Leaders

In addition to the above, seminars have been presented at several universities.

7. DISCOVERIES, INVENTIONS, AND PATENTS

All of the results achieved in this research have been or will be documented in papers in the archival literature. Preliminary work is being conducted to apply for a patent covering the basic technology used in the compiler used to secure survivability mechanisms. The preliminary title of the patent is: Defeating security attacks on computer software that are based on static analysis.

APPENDIX A

“Application Error Recovery in Critical Information Systems”

APPLICATION ERROR RECOVERY IN CRITICAL INFORMATION SYSTEMS

John C. Knight

Matthew C. Elder

*Department of Computer Science
University of Virginia*

*151 Engineer's Way, P.O. Box 400740
Charlottesville, VA 22904-4740*

August 25, 2000

knight@cs.virginia.edu

elder@cs.virginia.edu

(804) 982-2216

Abstract

Critical infrastructure applications provide services upon which society depends heavily; these applications are themselves dependent on distributed information systems for all aspects of their operation and so survivability of the information systems is an important issue. Fault tolerance is a key mechanism by which survivability can be achieved in these information systems. Fault tolerance consists of two primary stages, error detection and error recovery; in this paper we focus on application error recovery in these critical information systems. We outline a specification-based approach to error recovery that enables systematic structuring of error recovery specifications, an implementation partially synthesized from the formal specification, and various forms of static and run-time analysis. We present the RAPTOR specification notation for describing error recovery activities in the face of various faults, and we explore synthesis of implementation code using the Error Recovery Translator. We also describe a novel implementation architecture enabling error recovery in these systems and discuss issues in analysis.

APPLICATION ERROR RECOVERY IN CRITICAL INFORMATION SYSTEMS

1 Introduction

Our dependence on large infrastructure systems has increased to a point where the loss of the services that they provide is extremely disruptive [26], [27]. Infrastructures such as transportation, telecommunications, power distribution, and financial services are absolutely vital to the normal operation of society. Similarly, systems such as the Global Command and Control System (GCCS) are vital to defense operations. We refer to such applications as *critical infrastructure applications*.

These applications are themselves dependent on complex, distributed information systems, and all or most of the service provided by an infrastructure application can be lost quickly if faults arise in its information system. We refer to such information systems as *critical information systems*.

Having to deal with faults that might disrupt service in information systems leads to the notion of *survivability* [17]. Informally by survivability we mean the ability of the system to continue to provide service (possibly degraded) when serious traumas occur in the system or the operating environment. For example, when faults such as extensive hardware failure, software failure, operator error, or malicious attack occur, a critical subset of normal functionality or some alternative functionality might be needed to mitigate the consequences of the fault.

Survivability is a system *requirement*. The particular survivability requirements that must be met are described in a survivability specification that is a statement of the prioritized list of service levels that the system must provide and the associated probability for each service level being provided. There is no presumption about how survivability will be achieved in the notion of survivability itself. One essential aspect of system design is to ensure that systems are built to satisfy the probabilistic service requirements dictated by the survivability specification.

There are many mechanisms or strategies that can be employed to achieve survivability requirements in the face of faults in the system or changes in the system environment. The process of building a system in such a way that certain faults do not arise is *fault avoidance*. Building systems that are able to react in a requisite way to prescribed faults when they do arise is *fault tolerance* [2], [11]. In practice, the use of fault tolerance is essential in the types of heterogeneous distributed systems that underlie critical infrastructure applications. Fault tolerance breaks down into two general stages: first, detecting the effects of the fault, i.e., *error detection*, and second, dealing with the effects of the fault, i.e., *error recovery* [20].

In this document we summarize the issues involved in achieving error recovery in critical information systems. In Section 2, we describe some of the characteristics of the application programs with which we are concerned, critical information systems. The following section provides a more detailed description of the faults with which we are concerned and some fault tolerance definitions. Section 4 presents a motivating example, while the next section outlines an overview of our solution approach. A key component of our solution approach involves formal specification and synthesis of error recovery in these systems; Section 6 goes into great detail exploring

that aspect of the problem. Section 7 describes possible system architectures supporting error recovery, and after that we explore possible analyses that can be performed given formal specifications. Finally, we conclude with an overview of related work and a series of appendices outlining our work on this topic.

2 Critical Information Systems

The President's Commission on Critical Infrastructure Protection cited a variety of infrastructure application domains that have become exceedingly dependent upon their information systems for correct and efficient operation, including banking and finance, various transportation industries, electric power generation and distribution, and telecommunications [27]. Detailed descriptions of four of these applications can be found elsewhere [16].

The architecture of the information systems upon which critical infrastructure applications rely are tailored substantially to the services of the industries which they serve and influenced inevitably by cost-benefit trade-off's. For example, though these systems are typically distributed over a very wide (geographically dispersed) area with large numbers of nodes, the application dictates the sites and the distribution of nodes at those locations. Beyond this, however, there are a variety of similar characteristics possessed by these critical information systems across all application domains that are pertinent to achieving the requirement of enhanced survivability using new error recovery techniques:

- *Heterogeneous nodes.* Despite the large number of nodes in many of these systems, a small number of nodes are often far more critical to the functionality of the system than the remainder. This occurs because critical parts of the system's functionality are implemented on just one or a small number of nodes. Heterogeneity extends also to the hardware platforms, operating systems, application software, and even authoritative domains.

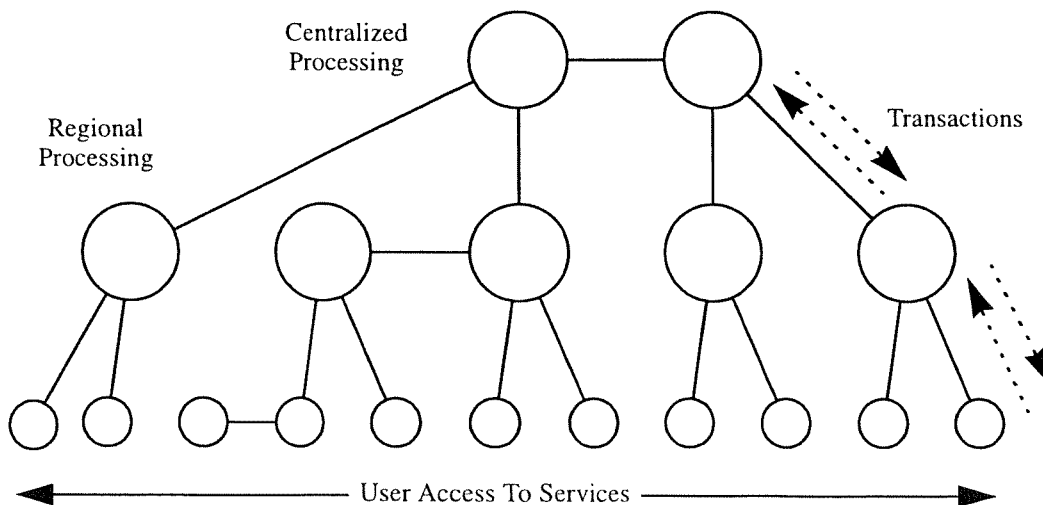


Figure 1. Example application network topology.

- *Composite functionality.* The service supplied to an end user is often attained by composing different functionality at different nodes. Thus, entirely different programs running on different nodes provide different services, and complete service can only be obtained when several subsystems cooperate and operate in some predefined sequence. This is quite unlike more familiar applications such as mail servers routing mail through the Internet. An example network topology for one of these applications is shown in Figure 1.
- *Stylized communication structures.* In a number of circumstances, critical infrastructure applications use dedicated, point-to-point links rather than fully-interconnected networks. Reasons for this approach include meeting application performance requirements, better security, and no requirement for full connectivity.
- *Performance requirements.* Some critical information systems, such as the financial payment system, have soft real-time constraints and throughput requirements (for checks cleared per second, for example), while others, such as parts of many transportation systems and many energy control systems, have hard real-time constraints. In some systems, performance requirements change with time as load or functionality changes—over a period of hours in financial systems or over a period of days or months in transportation systems, for example.
- *Security requirements.* Survivability is concerned with malicious attacks as well as failures caused by hardware and software faults. Given the importance of critical infrastructure applications, their information systems provide an attractive target to terrorists and other parties hostile to the nation intent on disrupting and sabotaging daily life. The deliberate faults exploited by security attacks are of significant concern to an error recovery strategy.
- *Extensive databases.* Infrastructure applications are concerned primarily with data. Many employ several extensive databases with different databases being located at different nodes and with most databases handling very large numbers of transactions.
- *COTS and legacy components.* For all the usual reasons, critical infrastructure applications utilize COTS components including hardware, operating systems, network protocols, database systems, and applications. In addition, these systems contain legacy components—custom-built software that has evolved with the system over many years.

The characteristics listed above are important, and most are likely to remain so in systems of the future. But the rate of introduction of new technology into these systems and the introduction of entirely new types of application is rapid, and these suggest that error recovery techniques must take into account the likely characteristics of future systems as well. We hypothesize that the following will be important architectural aspects of future infrastructure information systems:

- *Larger numbers of nodes.* The number of nodes in infrastructure networks is likely to increase dramatically as enhancements are made in functionality, performance, and user access. The effect of this on error recovery is considerable. In particular, it suggests that error recovery will have to be regional in the sense that different parts of the network will require different recovery strategies. It also suggests that the implementation effort involved in error recovery will be substantial because there are likely to be many regions and there will be many different anticipated faults, each of which might require different treatment.

- *Extensive, low-level redundancy.* As the cost of hardware continues to drop, more redundancy will be built into low-level components of systems. Examples include mirrored disks and redundant server groups. This will simplify error recovery in the case of low-level faults; however, catastrophic errors will still require sophisticated recovery strategies.
- *Packet-switched networks.* For many reasons, the Internet is becoming the network technology of choice in the construction of new systems, in spite of its inherent drawbacks (e.g., poor security and lack of performance guarantees). However, the transition to packet-switched networks, whether it be the current Internet or virtual-private networks implemented over some incarnation of the Internet, seems inevitable and impacts solution approaches for error recovery.

3 Faults and Fault Tolerance

We are concerned in this research with the need to tolerate faults that affect significant fractions of a network application, faults that we refer to as *non-local*. Thus, for example, a widespread power failure in which many application nodes are forced to terminate operation is a non-local fault. The complete failure of a single node upon which many other nodes depend would also have a significant non-local effect and would also be classified as a non-local fault.

Non-local faults have the important characteristic that they are usually *non-maskable*—that is, their effects are so extensive that normal system service cannot be continued with the resources that remain, even if the system includes extensive redundancy [7]. We are not concerned with faults at the level of a single hardware or software component. We refer to such faults as *local*, and we assume that all local faults are dealt with by some mechanism that masks their effects. Thus synchronized, replicated hardware components are assumed so that losses of single processors, storage devices, communications links, and so on are masked by hardware redundancy.

Non-local faults might affect a related subset of nodes in a network application leading to the idea that they can be *regional*. Thus, a fault affecting all the nodes in the banking system in a given city or state would be regional and dealing with such a fault might depend on the specific region that was affected. It is also likely that the elements of a non-local fault would manifest themselves over a period of time rather than instantly. This leads to the notion of *cascading* faults in which application components fail in some sequence over a possibly protracted period of time. Detecting such a situation and diagnosing the situation correctly is a significant challenge.

As mentioned previously, tolerating a fault requires first that the effects of the fault be detected, i.e., *error detection*, and second that the effects of the fault be dealt with, i.e., *error recovery* [20]. The mechanism that we employ to implement fault tolerance is a *survivability architecture* (discussed in detail elsewhere [33]). Both error detection and error recovery have to be defined precisely if a fault-tolerant system is to be built and operated correctly, and several issues arise in dealing with them.

3.1 Error Detection

Error detection for a non-local fault requires the collection of information about the state of the network application and subsequent analysis of the information. Analysis is required to permit a conclusion about the underlying fault to be made given a spectrum of specific information.

The key problem that arises in dealing with error detection in large distributed systems is

defining precisely what circumstances are of interest. Events will occur on a regular basis that are associated with faults that are either masked or of no interest. These events have to be filtered and incorporated accurately in the detection of errors of interest. The possibility of false positives, false negatives, and erroneous diagnosis is considerable. In a banking system, for example, it is likely to be the case that local power failures are masked routinely yet, if a series of local failures occurs in sequence, they are probably part of a widespread cascading failure that needs to be addressed either regionally or nationally.

3.2 Error Recovery

Error recovery for a fault whose effects cannot be masked requires that the application be reconfigured following error detection. The goal of reconfiguration is to effect changes such as terminating, modifying, or moving certain running applications, and starting new applications. In a banking application, for example, it might be necessary to terminate low priority services such as on-line customer enquiry, and modify some services, such as limiting electronic funds transfers to corporate customers.

Unless provision for reconfiguration is made in the design of the application, reconfiguration will be ad hoc at best and impossible at worst [15]. The provision for reconfiguration in the application design has to be quite extensive in practice for three reasons:

- The number of fault types is likely to be large and each might require different actions following error detection.
- It might be necessary to complete reconfiguration in bounded time so as to ensure that the replacement service is available in a timely manner.
- Reconfiguration itself must not introduce new security vulnerabilities.

Just what is required to permit application reconfiguration depends, in large measure, on the design of the application itself. Provision must be made in the application design to permit the service termination, initiation, and modification that is required by the specified fault-tolerant behavior.

4 Illustrative Example

To illustrate some of the issues that arise in implementing fault tolerance, consider an extremely simple example that is part of a hypothetical financial network application.

The system architecture, shown in Figure 2, consists of a 3-node network with one money-center bank (N1), two branch banks (N2 and N3), and three databases (DB), one attached to each node. There are two full-bandwidth communications links (L1 and L2) and two low-bandwidth backup links (I1 and I2). There is a low-bandwidth backup link between the two branch banks (I3). The intended functionality of this system is implement a small financial payments system, effecting value transfer between customer accounts. In a system free of faults, the branch banks provide customer access (check deposit facilities) and local information storage (customer accounts), while the money-center bank maintains branch bank asset management and switching facilities for check clearance.

The faults with which we would be concerned in a system of this type would be the loss of a

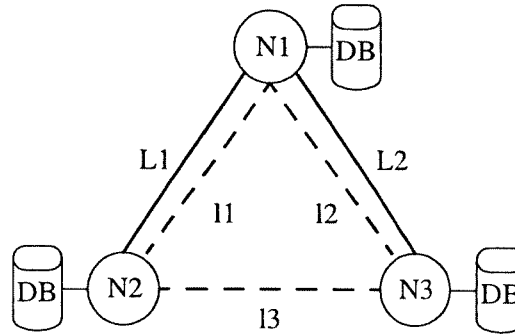


Figure 2. Example fault-tolerant distributed system.

computing node's hardware, the loss of an application program on a node, the loss of a database, the loss of a communications link, and so on. For each of these, it would be necessary first to define the fault and then, for each fault, document what the system is to do if the fault arises. In this example, losing the money-center bank would severely limit customer service since a branch bank would have to take over major services using link l3 for communication. Loss of either of the full-bandwidth communications links would also drastically cut service since communication would have to use a low-bandwidth link.

To implement a fault-tolerant system, the application must be constructed with facilities to tolerate the effects of the particular faults of concern. In the system architecture, the three low-bandwidth communications links provide alternate service in case of failures in the full-bandwidth communication links (L1 and L2) or the primary routing node (N1). The applications themselves must also provide alternate service in case of certain faults; for example, while the primary functionality of the money-center bank N1 is to route deposited checks for clearance and maintain the balances of each branch bank, additional services that can be provided include buffering of check requests for a failed branch bank or acceptance of checks for deposit if the branch banks can no longer provide this service. Similarly, the branch banks can be constructed to provide alternate service modes, such as the buffering of check requests in case of failure at the money-center bank, or buffering of low-priority check requests in case of failure of the full-bandwidth communications link.

Dealing with particular faults is only a small part of the problem. In practice, it will be necessary to deal with fault *sequences*, e.g., the loss of a communications link when the system has already experienced the loss of a node. In a large infrastructure network application, there are so many components that faults arising in sequence are a distinct possibility merely on stochastic grounds. However, cascading failures, sequenced terrorist attacks, or coordinated security attacks all yield fault sequences with faults that are potentially related, e.g., all links from a node are lost in some sequence or all the nodes in a geographic region are lost.

The various circumstances of interest can be described using a finite-state machine where each state is associated with a particular fault sequence. As well as enumeration of the states and associated state transitions associated with the faults that can arise, it is necessary to specify what has to be done on entry to each state in order to continue to provide service. Thus, application-

related actions have to be defined for each state transition, and the actions have to be tailored to both the initial state and the final state of the transition. Wide-area power failure has to be handled very differently if it occurs in a benign state versus when it occurs following a traumatic loss of computing equipment perhaps associated with a terrorist attack.

For this simple three-node example, we constructed a prototype error recovery specification to explore some of the issues involved in implementing a fault-tolerant system. The specification can be found in Appendix A. The first part of the specification is a description of the system itself. There are two aspects of the application that are described: the system architecture and the functionality (or services) provided by the system components. The description of the system architecture (Section A.1) consists of a listing of nodes (including attached databases) and connections. The functionality of each system component (Section A.2) is a listing of different services provided by each component of the system architecture, including alternate services available in case of various system failures. The system description is obviously informal, but it provides a basis for specification of the various system states that arise in the event of faults.

The second part of the specification is a description of the finite-state machine (Section A.3) and associated transitions (Section A.4) for the system and the fault sequences that are of concern. The initial state of the finite-state machine consists of a list of the services that are operational in the case of a fully-functional application. Transitions from this initial state are caused by faults, which manifest themselves as failures in one or more of the operational services. A high-level description of the fault that causes each transition is contained in the state description. In the finite-state machine transition section, a list of response activities is associated with each transition to attempt reconfiguration for the continued provision of service, as well as a list of activities to be undertaken upon repair of the fault.

In this simple three-node example, there are eleven components outlined in the specification of the system architecture that can fail, which would lead to faults at the system level. There are twenty-four services (some alternate) that those faults can affect; in the completely-operational initial state, there are ten services necessary to effect the fully-functional financial payments system. From the initial state there are eight possible faults causing transitions to other states. From those eight single-fault states, there are seventy-nine possible states should another fault occur. The finite-state machine in Appendix A only describes two sequential faults for this system. The complete finite-state machine enumerating all the states associated with the various possible fault sequences would have hundreds of states, even for a simple application system such as this.

The difficulties in achieving survivability, even in a system as simple as this example application, are clear. The first challenge lies in describing the relevant parts of the application, the system architecture and the functionality. Then, both the initial configuration and the changes to the system configuration in terms of that system description must be specified; the problem with state explosion and the impracticality of attempting to describe the finite state machine for a large network application is immediately obvious from the complications in this trivial system.

5 Solution Overview

As seen in the motivating example, to make a critical application fault tolerant, it is necessary to introduce mechanisms to recognize the errors of interest, maintain state information about the system to the extent that it affects error recovery, and define the required error recovery from all possible system states. The size of current and expected critical information systems, the variety and

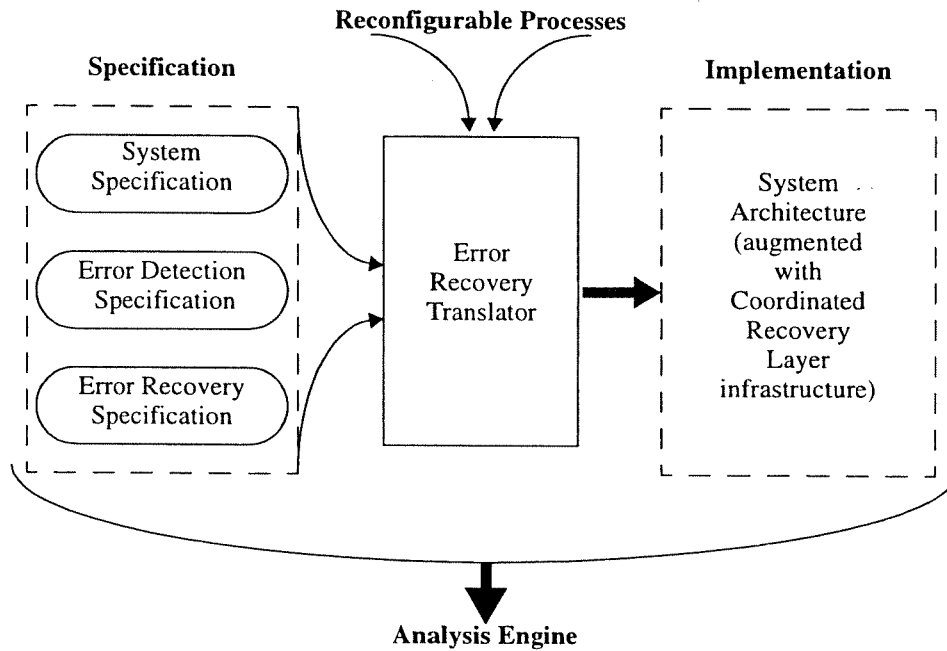


Figure 3. Error recovery methodology.

sophistication of the services they provide, and the complexity of the reconfiguration requirements mean that an approach to fault tolerance that depends upon traditional software development techniques is infeasible in all but the simplest cases. The likelihood is that future systems will involve at least tens of thousands of nodes, have to tolerate dozens, perhaps hundreds, of different types of fault, and have to support applications that provide very elaborate user services. Programming fault tolerance for such a system using conventional methods is quite impractical.

For these reasons, our solution approach is based on the use of a formal specification to describe the required error detection and error recovery, and the use of synthesis to generate the implementation from the formal specification. A formal specification describes the system, the faults that must be tolerated, and the application responses to those faults. The use of a formal specification notation enables a translator, the Error Recovery Translator, to synthesize portions of the implementation dealing with error recovery. In addition to the specification, input to the translator includes a special type of process—called a reconfigurable process—that supports the application reconfigurations described in the specification. The system architecture will consist of these reconfigurable processes, the synthesized code produced by the translator, and a support infrastructure called the Coordinated Recovery Layer that provides services to the reconfigurable processes. Finally, an analysis engine provides various static and run-time checking capabilities to ensure correctness of the error recovery activities.

An overview of our solution approach is shown in Figure 3. The major solution components and issues relating to each aspect of the solution are described in subsequent sections of this paper:

- **Specification and Synthesis**

In Section 6, we explore the issues involved in specifying and synthesizing error recovery. The specification is broken down into a variety of parts that define the different aspects of the problem as described previously. A unique element of the approach, however, is the use of a *multi-level* specification: the required error detection and error recovery specifications are written in terms of high-level abstract entities, defined in an intermediate specification that exports high-level entities but is itself written in terms of the low-level system components. The use of multiple specification levels for abstraction enables better control over the size and scope of these specifications. In addition, a synthesizer to process these formal specification notations has been built; the grammar for this Error Recovery Translator and the issues involved in generating code are presented in this section.

- **Implementation: Node and System Architecture**

In Section 7, we explore the issues involved in an implementation architecture to support error recovery. At the node level, each node must be constructed such that it supports reconfiguration and incorporates the generated code that implements reconfiguration. At the system level, coordination and control services must be provided to the reconfiguring nodes; a global entity called the Coordinated Recovery Layer provides these support services.

- **Analysis**

In Section 8, we explore the issues involved in analysis of the error recovery specifications and implementation. The use of formal specification permits various forms of analyses to check the error recovery algorithms: various forms of syntactic and semantic analyses are enabled by the formal specification notations, including invariant assertion checking. In addition, the implementation architecture supports various run-time checks to be performed.

6 Issues in Specification and Synthesis of Error Recovery

The first issue of concern in formally specifying error recovery is determining the components required in a specification language. Using the simple three-node system described in the motivating example, it can be seen that a finite-state machine must be constructed to specify the initial configuration of the system and any reconfigurations required to recover from system errors. This finite-state machine consists of all possible states the system could be in given the system errors that are of interest and should be handled, and the activities to undertake on transition from one system state to another. Two key aspects of the system must be described: the system architecture and the services that the nodes of the system provide.

As seen in the motivating example, even for a simple system an error recovery specification can become very large and unwieldy. Three observations can be made to deal with the specification size and state explosion problems:

- The specification notation must consist of multiple sub-specifications for describing the various components of the error recovery solution, e.g. the relevant system characteristics and the finite-state machine. These sub-specifications must be integrated to describe the overall error recovery solution, but the use of multiple sub-specifications enables different notations to be utilized and optimized for the particular aspect of the solution being addressed.
- The specification notation must be enhanced to accommodate larger numbers of nodes. One

way of achieving this would be to introduce and integrate some form of set-based notation to enable description and manipulation of large numbers of nodes simultaneously.

- The specification itself must be constructed in such a way as to keep it manageable: for example, portions of the system can be abstracted and consolidated into single objects in the specification, thus ensuring that the specification deals with and manipulates small numbers of objects regardless of how many actual nodes there are in the system.

Using the observations from the exercise in the motivating example, the first-generation RAPTOR specification notation was devised to specify error recovery in critical information systems.

6.1 The RAPTOR Specification Notation

The first-generation RAPTOR specification notation involves four major sub-specifications:

- **Error Detection Specification (EDS)**
The error-detection specification defines the overall systems states that are associated with the various faults of interest.
- **Error Recovery Specification (ERS)**
The error-recovery specification defines the necessary state changes from any acceptable system reconfiguration to any other in terms of topology, functionality, and geometry (assignment of services to nodes).
- **System Architecture Specification (SAS)**
The system architecture specification describes the topology of the system and platform including the computing nodes, the communications links, and detailed parametric information for key characteristics. For example, nodes are named and described additionally with node type, hardware details, operating system, software versions, and so on. Links are specified with connection type and bandwidth capabilities.
- **Service-Platform Mapping Specification (SPMS)**
The service-platform mapping specification relates the names of programs to the node names described in the SAS. The program descriptions in the SPMS include the services that each program provides, including alternate and degraded service modes.

Given the number of states that a large distributed system could enter as a result of the manifestation of a sequence of faults, it is clear that some form of accumulation of states or other simplification must be imposed if an approach even to specification of fault tolerance is to be tractable. The key to this simplification lies in the fact that many nodes in large networks, even those providing critical infrastructure service, do not need to be distinguished for purposes of fault tolerance. In the banking application, for example, it is clear that the loss of computing service at any single branch is both largely insignificant and largely independent of which branch is involved. Conversely, the loss of even one of the main Federal Reserve computing or communications centers would impede the financial system dramatically—some nodes are much more critical than others. However, the loss of 10,000 branch banks (for example because of a common-mode software error) would be extremely serious—even non-critical nodes have an impact if sufficient of them are lost at the same time.

To cope with the required accumulation of states, the overall specification is made two-level,

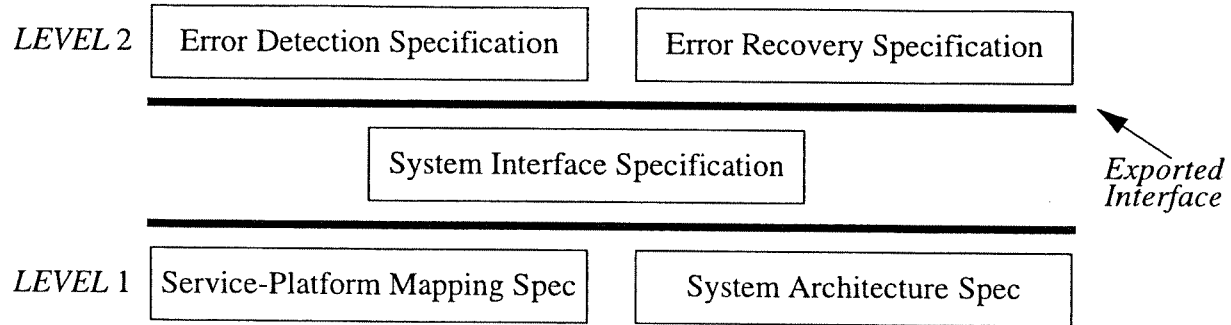


Figure 4. Two-level specification structure.

and we add a fifth element to the specification approach. The SAS and the SPMS are declarative specifications, and in practice the content of these specifications are databases of facts about the system architecture and configuration. The EDS and ERS are both algorithmic specifications—they describe algorithms that have to be executed to perform error detection and recovery respectively. In principle, these algorithms can be written using the information contained in the SAS and SPMS. But it is precisely this approach that leads to the state explosion in specification. The SAS and SPMS describing these systems are just too big.

The fifth element is the *system interface specification* (SIS). This is a specification that defines major system objects in terms of the lower-level entities contained in the SAS and SPMS. These objects are exported from the SAS and SPMS and become the objects with which the EDS and ERS are written. This structure is shown in Figure 4.

The overall structure of the ERS is that of a (traditional) finite-state machine that characterizes fault conditions as states (defined using sets) and the requisite responses to each fault are associated with state transitions. The fault conditions of concern for a given system are declared and described in the EDS. Arcs in the ERS finite-state machine are labeled with these fault conditions and show the state transitions for each fault from every relevant state. The actions associated with any given transition are in the ERS and are extensive because each action is essentially a high-level program that implements the error-recovery component of the full system survivability specification. The complete system-survivability specification documents the different states (system environments) that the system can be in, including the errors that will be detected and handled. In the RAPTOR specification notation's ERS a notational construct was designed to describe the finite-state machine of the system through all system errors. The notational construct for the finite-state machine enables brute-force description of all possible relevant failures in all possible states as well as the responses to those failures.

In summary, a first-generation RAPTOR specification consists of five subsections that correspond to the five sub-specifications outlined above. An example fragment of a RAPTOR specification is shown in Figure 5. This example is incomplete and uses comments for simplicity but it illustrates some of the material needed to define a wide-area coordinated security attack on the banking system and a hypothetical response that might be required.

```

RAPTOR-SAS:                -- System architecture specification
    -- Use Z-like given sets for illustration
    -- [retail_banks], [regional_banks], [federal_reserve]

RAPTOR-PMS:                -- System platform mapping specification
    forall i: retail_banks[i]    -> customer_service;
                                   local_payment;
    forall i: regional_banks[i]   -> customer_account_management;
                                   regional_payment;
    forall i: federal_reserve[i]  -> member_bank_account_management;
                                   national_payment;

RAPTOR-SIS:                -- System interface specification
    Events:
        -- attack      == intrusion_detection_alarm triggered;
    Objects:
        branch_banks  == {i : bank | i memberof retail_banks}
        district_banks == {i : bank | i memberof regional_banks}
        central_banks == {i : bank | i memberof federal_reserve}

RAPTOR-EDS:                -- Error detection specification
    coordinated_attack == card(attack(branch_banks)) > 1000
                        OR card(attack(district_banks)) > 3
                        OR card(attack(central_banks)) > 1;
    -- define this attack to be more than 10 branches or
    -- more than 3 money center or more than one Federal reserve
    -- bank detects an intrusion (via an intrusion detection system)

RAPTOR-ERS:                -- Error recovery specification
    On coordinated_attack:
        branch_banks    -> customer_service.terminate;
                        local_payment.terminate;
                        local_enquiry.start;
        money_center_banks -> customer_account_management.terminate;
                        regional_payment.terminate;
                        commercial_account_management.start;
        federal_reserve  -> member_bank_account_management.terminate;
                        national_payment.limit;

```

Figure 5. Skeleton RAPTOR specification.

6.2 RAPTOR Parser: Error Recovery Translator

The first-generation RAPTOR specification language has a parser, constructed using LEX and YACC. The parser processes all five sub-specification notations and generates C++ code for the actuators of the nodes in the run-time system. This Error Recovery Translator is constructed from a grammar with 41 productions and 31 tokens (presented in Appendix B). It contains facilities for simple set enumeration and composition, boolean logic, and quantifiers. The translator generates code on a per-node basis for all nodes declared in the System Architecture Specification.

The grammar for the RAPTOR language parses the RAPTOR sub-specification languages in a particular order: SAS, SPMS, SIS, EDS, and ERS. In the SAS, first all of the nodes are declared,

then the node types, properties, and events are declared. This is followed by a list of propositions stating facts about the nodes, assigning types and properties to nodes. In the SPMS, service names are assigned to the node types. In the SIS, the sets to be used are declared and then the sets are enumerated, using either explicit set declaration or set composition across node type or property. In the EDS, the system errors are declared and then described in terms of events in individual nodes or quantified across sets. Finally, in the ERS, the finite-state machine is described using the errors declared in the EDS, and the actions to be taken on state transitions are specified.

6.3 Observations and Analysis

To explore the issues in the Error Recovery Translator, a sample specification of a 100-node financial payments application was constructed. Actuator code to effect reconfiguration for error recovery was generated for the nodes in the system and integrated into the system implementation. The specification for the 100-node system can be found in Appendix C. This exercise yielded many observations related both to the notation and translator:

- The System Architecture Specification for defining all the nodes in the system and their pertinent characteristics is cumbersome for describing a system of any non-trivial size in any detail. Because this information most certainly exists in databases already, constructing an input filter to process that information directly would be a simpler way to define the architecture and characteristics of the system.
- The Service-Platform Mapping Specification uses a simple naming mechanism for defining services; this should be augmented with more formal definition of the functionality provided by each service.
- Errors defined in the Error Detection Specification should be parameterized when used in the Error Recovery Specification. This will allow errors arising at different nodes of the same type to be distinguished, if necessary, while not increasing the size and complexity of the Error Recovery Specification.

One particular specification problem that was explored in the motivating example was the state explosion problem in the finite-state machine. The first-generation RAPTOR specification notation does not address this problem explicitly in the description of the finite-state machine in the ERS. However, there are a number of approaches that can be employed to control the state explosion problem:

- The number of possible system states must be restricted - in systems this large and complex, some forms of abstraction will have to be employed to keep the number of possible system states tractable.
- The number of sequential failures that can be tolerated must be restricted - handling large numbers of sequential failures in the general case cannot be achieved in a specification of reasonable size.
- If a system can be constructed such that sequential failures are handled independently of the order or occurrence of previous failures, then the specification becomes very simple: only the number of different failures that can occur in the system must be specified and handled.

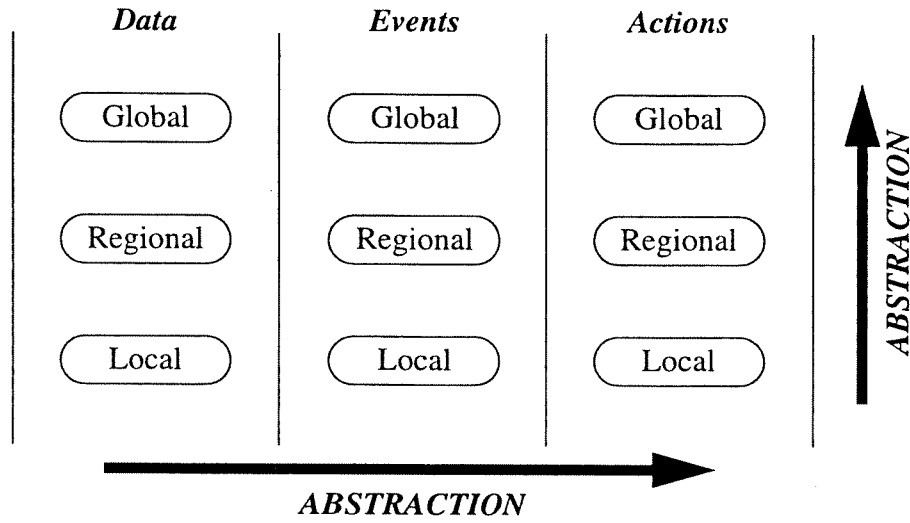


Figure 6. Two directions of abstraction.

6.4 Concepts for the Second-Generation RAPTOR Specification Notation

The consideration of issues in the finite-state machine description and state explosion problem in these critical, large-scale systems points towards the exploration of abstraction and hierarchy to help control state. These systems are large and geographically distributed; thus, there will be a notion of regional control and recovery. This characteristic lends itself to hierarchical control of error detection and recovery. This introduces abstraction in a complementary manner, or direction, to the abstraction discussed previously. In the previous form of abstraction, information or data is abstracted at lower levels of the system, then aggregated into sets (objects) at the system level and events (system errors) occurring with respect to those objects; finally, those events cause actions to occur in response, typically reconfiguration to recover from those system errors.

Hierarchy introduces abstraction in a second direction though (see Figure 6). Data, events, and actions can be occurring independently at different levels of the system hierarchy, such as at the local level, regional level, and global level. It is then possible for data, events, and actions at the local level to be passed up to the regional level, and so on to the global level.

This second abstraction mechanism can be used to help control the state explosion problem at the system level. It should be possible to construct finite-state machines at the node or local level to control error detection and error recovery internally, and only pass data, events, or actions up to the regional level in circumstances where the situation calls for that. Likewise, the system or global level state machine can be simplified by handling regional data, events, and actions at the lower level whenever possible.

7 Issues in a System Architecture for Error Recovery

A critical information system that supports error recovery in the manner described above must

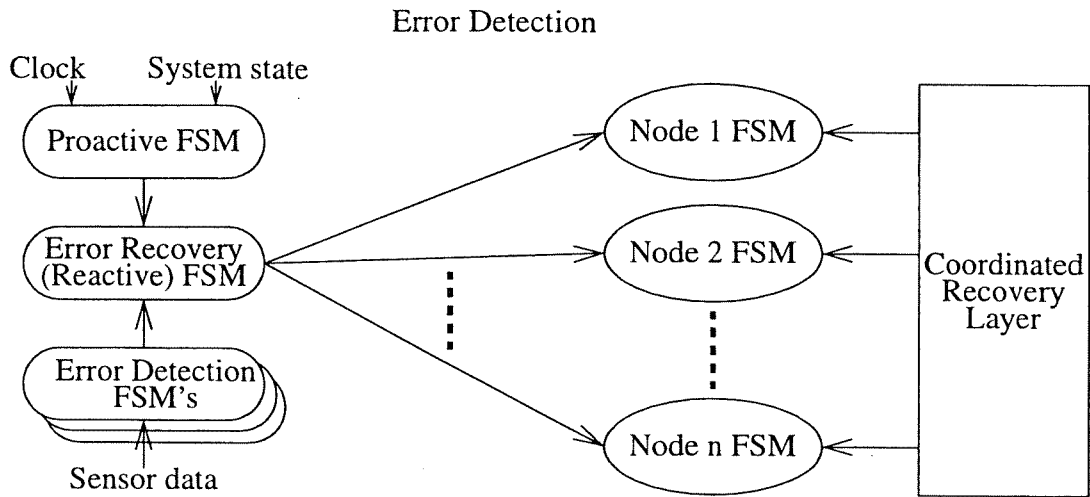


Figure 7. System architecture for error recovery.

have architectural support. In particular, both at the node level and the system level there must be mechanisms in place to effect reconfiguration. At the node level, the synthesized code for the actuator generated by the Error Recovery Translator must be integrated with the existing application code. At the system level, support for synchronization and coordination must be provided for reconfiguring nodes. The Coordinated Recovery Layer is a system-level construct that provides these services to the nodes.

7.1 Observations and Analysis

In terms of the implementation of the system architecture, a key design question concerns the location of the services provided by the Coordinated Recovery Layer. For example, one important coordination service is coordinated commitment by all nodes regarding reconfiguration actions: all of the nodes must receive the reconfiguration command and commit to that action at the same time.

One possible mechanism for implementing coordinated commitment is the two-phase commit protocol. Clearly, the nodes will have to participate in this protocol, so the services that are provided to the nodes by the Coordinated Recovery Layer really must exist within the node applications. However, the command to reconfigure will come from a separate entity in the survivability architecture, the control system. Therefore, one possible implementation would be to make the control system the coordinator in the two-phase commit protocol, thus locating that particular functionality of the Coordinated Recovery Layer in the control system. A key design question is whether all such services of the Coordinated Recovery Layer will exist in the control system and nodes (thus reducing the notion of the Coordinated Recovery Layer to a set of services provided in a library), or are there other services that must exist in a separate entity in the system that will be called the Coordinated Recovery Layer.

7.2 Future Directions

The second-generation RAPTOR specification notation points towards an implementation architecture similar to that pictured in Figure 7. The introduction of hierarchic finite-state machines for events, data, and actions requires finite-state machines at each node to control local error recovery. Sensor data from the various nodes will still feed into an error detection component to determine regional and global errors, and any faults recognized at those levels will initiate transitions at the higher-level error recovery finite-state machine(s). Finally, a finite-state machine for proactive error recovery activities must be introduced to control operations such as coordinated checkpointing.

8 Issues in Analysis of Error Recovery

The use of a formal notation for specification of error recovery permits various forms of analysis to be performed for correctness of the algorithms. The use of a parser checks the syntax of the specifications automatically to avoid simple errors of form. In addition, tools exist to analyze finite state machines for such characteristics as states with no exiting transitions.

More importantly though, semantic analysis can be performed on the specifications to ensure correctness of the error recovery. For example, the error recovery specification can be augmented with assertions to denote the fundamental processing or service to be provided during each state. Then the specification can be analyzed to prove that the required level of service is provided irrespective of the sequence of faults.

The system architecture also permits analysis of the run-time system. The control system continually monitors various nodes; the state information being collected from these nodes can include run-time assertions that check the provision of various levels of service.

9 Related Work

9.1 Fault-Tolerant Systems

Many system-level approaches exist that provide various subsets of abstractions and services. In this subsection we survey some of the existing work on fault-tolerant system architectures.

9.1.1. Cristian/Advanced Automation System. Cristian provided a survey of the issues involved in providing fault-tolerant distributed systems [8]. He presented two requirements for a fault-tolerant system: 1) mask failures when possible, and 2) ensure clearly specified failure semantics when masking is not possible. The majority of his work, however, dealt with the masking of failures.

An instantiation of Cristian's fault tolerance concepts was used in the replacement Air Traffic Control (ATC) system, called the Advanced Automation System (AAS). The AAS utilized Cristian's fault-tolerant architecture [9]. Cristian described the primary requirement of the air traffic control system as ultra-high availability and stated that the approach taken was to design a system that can automatically mask multiple concurrent component failures.

The air traffic control system described by Cristian handled relatively low-level failures: redundancy of components was utilized and managed in order to mask these faults. Cristian structured the fault-tolerant architecture using a "depends-on" hierarchy, and modelled the system in

terms of servers, services, and a “uses” relation. Redundancy was used to mask both hardware and software failures at the highest level of abstraction, the application level. Redundancy was managed by application software server groups [9].

9.1.2. Birman/ISIS, Horus, and Ensemble. A work similar to that of Cristian is the “process-group-based computing model” presented by Birman. Birman introduced a toolkit called ISIS that contained system support for process group membership, communication, and synchronization. ISIS balanced trade-off’s in closely synchronized distributed execution (which offers easy understanding) and asynchronous execution (which achieves better performance through pipelined communication) by providing the virtual synchrony approach to group communication. ISIS facilitated group-based programming by providing a software infrastructure to support process group abstractions. Both Birman and Cristian’s work addressed a “process-group-based computing model,” though Cristian’s AAS also provided strong real-time guarantees made possible by an environment with strict timing properties [5].

Work on ISIS proceeded in subsequent years resulting in another group communications system, Horus. The primary benefit of Horus over ISIS was a flexible communications architecture that can be varied at runtime to match the changing requirements of the application and environment. Horus achieved this flexibility using a layered protocol architecture in which each module is responsible for a particular service [35]. Horus also worked with a system called Electra, which provided a CORBA-compliant interface to the process group abstraction in Horus [21]. Another system that built on top of Electra and Horus together, Piranha, provided high availability by supporting application monitoring and management facilities [22].

Horus was succeeded by a new tool for building adaptive distributed programs, Ensemble. Ensemble further enabled application adaptation through a stackable protocol architecture as well as system support for protocol switching. Performance improvements were also provided in Ensemble through protocol optimization and code transformations [36].

An interesting note on ISIS, Horus, and Ensemble was that all three acknowledged the security threats to the process group architecture and each incorporated a security architecture into its system [29], [30], [31].

9.1.3. Other system-level approaches. Another example of fault tolerance that focuses on communication abstractions is the work of Schlichting *et al.* The result of this work is a system called Coyote that supports configurable communication protocol stacks. The goals are similar to that of Horus and Ensemble, but Coyote generalizes the composition of microprotocol modules allowing non-hierarchical composition (Horus and Ensemble only support hierarchical composition). In addition, Horus and Ensemble are focusing primarily on group communication services while Coyote supports a variety of high-level network protocols [4].

Many of the systems mentioned above focus on communication infrastructure and protocols for providing fault tolerance; another approach focuses on transactions in distributed systems as the primary primitive for providing fault tolerance. One of the early systems supporting transactions was Argus, developed at MIT. Argus was a programming language and support system that defined transactions on software modules, ensuring persistence and recoverability [6].

Another transaction-based system, Arjuna, was developed at the University of Newcastle upon Tyne. Arjuna is an object-oriented programming system that provides atomic actions on objects using C++ classes [32]. The atomic actions ensure that all operations support the properties of serializability, failure atomicity, and permanence of effect.

9.1.4. Discussion. A common thread through the approach taken to fault tolerance in all of these systems is that faults are masked; each of these systems attempts to provide transparent masking of failures when a fault arises. Masking requires redundancy, and not all faults can be masked because it is not possible to build enough redundancy into a system to accommodate all faults in that way. Therefore, there will be a class of faults that these approaches to fault tolerance cannot handle because there is insufficient redundancy to tolerate them by masking.

Another interesting note is that these approaches tend to be communication-oriented. This is easily understandable - fault tolerance in general is dependent on redundancy, and one key to managing redundancy is maintaining consistent views of the state across all redundant entities. Supporting such a requirement within the communications framework—building guarantees into that framework—is a common approach to providing fault tolerance, but communications is not the only aspect of the system that must be addressed for a comprehensive error recovery strategy.

Finally, the scale of these systems tends not to be on the order of critical information systems.

9.2 Fault Tolerance in Wide-area Network Systems

Fault tolerance is typically applied to relatively small-scale systems, dealing with single processor failures and limited redundancy. Critical information systems are many orders of magnitude larger than the distributed systems that most of the previous work has addressed. There are, however, a few research efforts addressing fault tolerance in large-scale, wide-area network systems.

In the WAFT project, Marzullo and Alvisi are concerned with the construction of fault-tolerant applications in wide-area networks. Experimental work has been done on the Nile system, a distributed computing solution for a high-energy physics project. The primary goal of the WAFT project is to adapt replication strategies for large-scale distributed applications with dynamic (unpredictable) communication properties and a requirement to withstand security attacks. Nile was implemented on top of CORBA in C++ and Java. The thrust of the work thus far is that active replication is too expensive and often unnecessary for these wide-area network applications; Marzullo and Alvisi are looking to provide support for passive replication in a toolkit [1].

The Eternal system, developed by Melliar-Smith and Moser, is middleware that operates in a CORBA environment, below a CORBA ORB but on top of their Totem group communication system. The primary goal is to provide transparent fault tolerance to users [25].

Babaoglu and Schiper are addressing problems with scaling of conventional group technology. Their approach for providing fault tolerance in large-scale distributed systems consists of distinguishing between different roles or levels for group membership and providing different service guarantees to each level [3].

9.2.1. Discussion. While the approaches discussed in this section accommodate systems of a larger scale, many of the concerns raised previously still apply. These efforts still attempt to mask faults using redundancy and they are still primarily communications-oriented. There is still a class of faults that cannot be handled because there is insufficient redundancy.

9.3 Reconfigurable Distributed Systems

Given the body of literature on fault tolerance and the different services being provided at each abstraction layer, many types of faults can be handled. However, the most serious fault—the catastrophic, non-maskable fault—is not addressed by the previous related work. The previous

approaches rely on having sufficient redundancy to cope with the fault and mask it; there are always going to be classes of faults for which this is not possible. For these faults, reconfiguration of the existing services on the remaining platform is required.

Considerable work has been done on reconfigurable distributed systems. Some of the work deals with reconfiguration for the purposes of evolution, as in the CONIC system, and, while this work is relevant, it is not directly applicable because it is concerned with reconfiguration that derives from the need to upgrade rather than cope with major faults. Less work has been done on reconfiguration for the purposes of fault tolerance. Both types of research are discussed in this section.

9.3.1. Reconfiguration supporting system evolution. The initial context of the work by Kramer and Magee was dynamic configuration for distributed systems, incrementally integrating and upgrading components for system evolution. CONIC, a language and distributed support system, was developed to support dynamic configuration. The language enabled specification of system configuration as well as change specifications, then the support system provided configuration tools to build the system and manage the configuration [18].

More recently, they have modelled a distributed system in terms of processes and connections, each process abstracted down to a state machine and passing messages to other processes (nodes) using the connections. One relevant finding of this work is that components must migrate to a “quiescent state” before reconfiguration to ensure consistency through the reconfiguration; basically, a quiescent state entailed not being involved in any transactions. The focus remained on the incremental changes to a distributed system configuration for evolutionary purposes [19].

The successor to CONIC, Darwin, is a configuration language that separates program structure from algorithmic behavior [24]. Darwin utilizes a component- or object-based approach to system structure in which components encapsulate behavior behind a well-defined interface. Darwin is a declarative binding language that enables distributed programs to be constructed from hierarchically-structured specifications of component instances and their interconnections [23].

9.3.2. Reconfiguration supporting fault tolerance. Purtilo developed the Polyolith Software Bus, a software interconnection system that provides a module interconnection language and interfacing facilities (software toolbus). Basically, Polyolith encapsulates all of the interfacing details for an application, where all software components communicate with each other through the interfaces provided by the Polyolith software bus [28].

Hofmeister extended Purtilo’s work by building additional primitives into Polyolith for support of reconfigurable applications. Hofmeister studied the types of reconfigurations that are possible within applications and the requirements for supporting reconfiguration. Hofmeister leveraged heavily off of Polyolith’s interfacing and message-passing facilities in order to ensure state consistency during reconfiguration [13].

Welch and Purtilo have extended Hofmeister’s work in a particular application domain, Distributed Virtual Environments. They utilized Polyolith and its reconfiguration extensions in a toolkit that helps to guide the programmer in deciding on proper reconfigurations and implementations for these simulation applications [37].

9.3.3. Discussion. The research on reconfiguration for the purposes of evolution is interesting but of course does not work on the same time scale as required for survivability in critical information systems. Critical information systems have performance requirements that must still be met by an

error recovery mechanism; reconfiguration during evolution is not concerned with performance in general.

Reconfiguration for the purposes of fault tolerance, however, does concern itself with performance. However, the existing solutions do not accommodate systems on the scale of critical information systems. One might argue as well that these research efforts do not handle the complexity and heterogeneity of critical information systems.

10 Conclusions

Fault tolerance in critical information systems is essential because the services that such systems provide are crucial. In attempting to deal with faults in such systems, it becomes clear immediately that the complexity of the fault-tolerance mechanism itself could be a serious liability for the system. The number of system states and the number of possible faults are such that the creation of a fault-tolerant system using typical hand-crafted development is infeasible.

We have developed a specification-based approach that deals with the problem by reducing the problem to the creation of a formal specification from which an implementation is synthesized. The complexity of the specification itself is reduced significantly by using a two-level structure. We note that the implementation issues which arise in the approach that we have described are *very* significant but are not addressed in this paper.

The overall approach permits fault tolerance to be introduced into network applications in a manageable way. The price of achieving this is a loss in flexibility. The utility of the approach is presently under investigation as are the details of implementation.

11 Acknowledgments

This effort sponsored in part by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0314. The U.S. Government is authorized to reproduce and distribute reprints for governmental purpose notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

12 References

- [1] Alvisi, L. and K. Marzullo. "WAFT: Support for Fault-Tolerance in Wide-Area Object Oriented Systems," Proceedings of the 2nd Information Survivability Workshop, IEEE Computer Society Press, Los Alamitos, CA, October 1998, pp. 5-10.
- [2] Anderson, T. and P. Lee. Fault Tolerance: Principles and Practice. Prentice Hall, Englewood Cliffs, NJ, 1981.
- [3] Babaoglu, O. and A. Schiper. "On Group Communication in Large-Scale Distributed Systems," ACM Operating Systems Review, Vol. 29 No. 1, January 1995, pp. 62-67.
- [4] Bhatti, N., M. Hiltunen, R. Schlichting, and W. Chiu. "Coyote: A System for Constructing Fine-Grain Configurable Communication Services," ACM Transactions on Computer Systems, Vol. 16 No. 4, November 1998, pp. 321-366.

- [5] Birman, K. "The Process Group Approach to Reliable Distributed Computing," *Communications of the ACM*, Vol. 36 No. 12, December 1993, pp. 37-53 and 103.
- [6] Birman, K. Building Secure and Reliable Network Applications. Manning, Greenwich, CT, 1996.
- [7] Cowan, C., L. Delcambre, A. Le Meur, L. Liu, D. Maier, D. McNamee, M. Miller, C. Pu, P. Wagle, and J. Walpole. "Adaptation Space: Surviving Non-Maskable Failures," Technical Report 98-013, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, May 1998.
- [8] Cristian, F. "Understanding Fault-Tolerant Distributed Systems," *Communications of the ACM*, Vol. 34 No. 2, February 1991, pp. 56-78.
- [9] Cristian, F., B. Dancey, and J. Dehn. "Fault-Tolerance in Air Traffic Control Systems," *ACM Transactions on Computer Systems*, Vol. 14 No. 3, August 1996, pp. 265-286.
- [10] Ellison, B., D. Fisher, R. Linger, H. Lipson, T. Longstaff, and N. Mead. "Survivable Network Systems: An Emerging Discipline," Technical Report CMU/SEI-97-TR-013, Software Engineering Institute, Carnegie Mellon University, November 1997.
- [11] Gartner, Felix C. "Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments," *ACM Computing Surveys*, Vol. 31 No. 1, March 1999, pp. 1-26.
- [12] Hofmeister, C., E. White, and J. Purtilo. "Surgeon: A Packager for Dynamically Reconfigurable Distributed Applications," *Software Engineering Journal*, Vol. 8 No. 2, March 1993, pp. 95-101.
- [13] Hofmeister, C. "Dynamic Reconfiguration of Distributed Applications," Ph.D. Dissertation, Technical Report CS-TR-3210, Department of Computer Science, University of Maryland, January 1994.
- [14] Jalote, P. Fault Tolerance in Distributed Systems. Prentice Hall, Englewood Cliffs, NJ, 1994.
- [15] Knight, J., M. Elder, and X. Du. "Error Recovery in Critical Infrastructure Systems," *Proceedings of Computer Security, Dependability, and Assurance '98*, IEEE Computer Society Press, Los Alamitos, CA, 1999, pp. 49-71.
- [16] Knight, J., M. Elder, J. Flinn, and P. Marx. "Summaries of Three Critical Infrastructure Systems," Technical Report CS-97-27, Department of Computer Science, University of Virginia, November 1997.
- [17] Knight, J. and K. Sullivan. "Towards a Definition of Survivability," *Proceedings of the Third Information Survivability Workshop*, October 2000.
- [18] Kramer, J. and J. Magee. "Dynamic Configuration for Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. SE-11 No. 4, April 1985, pp. 424-436.
- [19] Kramer, J. and J. Magee. "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Transactions on Software Engineering*, Vol. 16 No. 11, November 1990, pp. 1293-1306.
- [20] Laprie, Jean-Claude. "Dependable Computing and Fault Tolerance: Concepts and Terminology," *Digest of Papers FTCS-15: 15th International Symposium on Fault-Tolerant Computing*, pp. 2-11, 1985.
- [21] Maffeis, S. "Electra - Making Distributed Programs Object-Oriented," Technical Report 93-

- 17, Department of Computer Science, University of Zurich, April 1993.
- [22]Maffeis, S. "Piranha: A CORBA Tool For High Availability," IEEE Computer, Vol. 30 No. 4, April 1997, pp. 59-66.
 - [23]Magee, J., N. Dulay, S. Eisenbach, and J. Kramer. "Specifying Distributed Software Architectures," Lecture Notes in Computer Science, Vol. 989, September 1995, pp. 137-153.
 - [24]Magee, J. and J. Kramer. "Darwin: An Architectural Description Language," <http://www-dse.doc.ic.ac.uk/research/darwin/darwin.html>, 1998.
 - [25]Melliard-Smith, P. and L. Moser. "Surviving Network Partitioning," IEEE Computer, Vol. 31 No. 3, March 1998, pp. 62-68.
 - [26]Office of the Undersecretary of Defense for Acquisition and Technology. "Report of the Defense Science Board Task Force on Information Warfare - Defense (IW-D)," November 1996.
 - [27]President's Commission on Critical Infrastructure Protection. "Critical Foundations: Protecting America's Infrastructures The Report of the President's Commission on Critical Infrastructure Protection," United States Government Printing Office (GPO), No. 040-000-00699-1, October 1997.
 - [28]Purtilo, J. "The POLYLITH Software Bus," ACM Transactions on Programming Languages and Systems, Vol. 16 No. 1, January 1994, pp. 151-174.
 - [29]Reiter, M., K. Birman, and L. Gong. "Integrating Security in a Group Oriented Distributed System," Proceedings of the 1992 IEEE Symposium on Research in Security and Privacy, IEEE Computer Society Press, Los Alamitos, CA, May 1992, pp. 18-32.
 - [30]Reiter, M., K. Birman, and R. van Renesse. "A Security Architecture for Fault-Tolerant Systems," ACM Transactions on Computer Systems, Vol. 12 No. 4, November 1994, pp. 340-371.
 - [31]Rodeh, O., K. Birman, M. Hayden, Z. Xiao, and D. Dolev. "Ensemble Security," Technical Report TR98-1703, Department of Computer Science, Cornell University, September 1998.
 - [32]Shrivastava, S., G. Dixon, G. Parrington. "An Overview of the Arjuna Distributed Programming System," IEEE Software, Vol. 8 No. 1, January 1991, pp. 66-73.
 - [33]Sullivan, K., J. Knight, X. Du, and S. Geist. "Information Survivability Control Systems," Proceedings of the 21st International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, May 1999, pp. 184-192.
 - [34]Summers, B. The Payment System: Design, Management, and Supervision. International Monetary Fund, Washington, DC, 1994.
 - [35]van Renesse, R., K. Birman, and S. Maffeis. "Horus: A Flexible Group Communications System," Communications of the ACM, Vol. 39 No. 4, April 1996, pp. 76-83.
 - [36]van Renesse, R., K. Birman, M. Hayden, A. Vaysburd, and D. Karr. "Building Adaptive Systems Using Ensemble," Technical Report TR97-1638, Department of Computer Science, Cornell University, July 1997.
 - [37]Welch, D. "Building Self-Reconfiguring Distributed Systems using Compensating Reconfiguration," Proceedings of the 4th International Conference on Configurable Distributed Systems, IEEE Computer Society Press, Los Alamitos, CA, May 1998.

Appendix A

3-Node Example Prototype System Specification

This example system is a small-scale financial payments application, effecting value transfer between customer accounts. The system consists of three nodes, two branch banks and one money-center bank, and various connections between these nodes. The branch banks are intended to provide customer access (check deposit facilities) and local information storage (customer accounts), while the money-center bank is intended to track branch bank asset balances and route checks for clearance between the two branch banks.

A.1 System Architecture Specification

The system architecture is pictured in Figure 8.

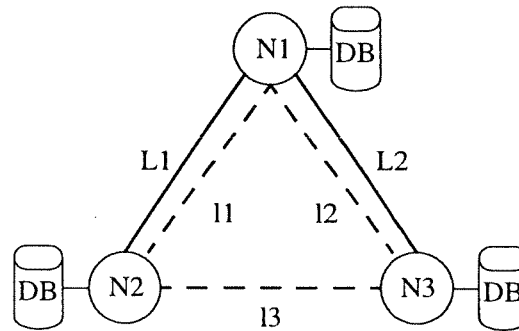


Figure 8. Example financial payments application.

The system consists of a 3 nodes:

- N1: money-center bank (MCB)
- N2: a branch bank (BB)
- N3: branch bank (BB)

There is a database (DB) attached to each node:

- DB(N1)
- DB(N2)
- DB(N3)

There are five total connections (primary and backup) between the various nodes:

- L1: full-bandwidth link between N1 and N2
- L2: full-bandwidth link between N1 and N3
- l1: low-bandwidth backup link between N1 and N2
- l2: low-bandwidth backup link between N1 and N3
- l3: full-bandwidth backup link between the two branch banks, N2 and N3

A.2 System Functionality Specification

In this section, the services that each node provides will be specified (named), with a brief description of each service.

The money center bank N1 provides the following services:

- MCB1: Route requests
- MCB2: Maintain branch total balances (DB service)
- MCB3: Buffer requests for a branch bank (Alternate)
- MCB4: Send buffered requests (Alternate)
- MCB5: Accept requests from customers (Alternate)

The branch banks N2 and N3 provide the following services:

- BB1: Accept requests from customers, routing to other branch bank if necessary
- BB2: Accept requests from other branch banks
- BB3: Process requests, maintaining customer balances (DB service)
- BB4: Buffer requests to pass up (Alternate)
- BB5: Send buffered requests (Alternate)
- BB6: Send high-priority requests, queue others (Alternate)
- BB7: Pass requests directly to branch bank (Alternate)

The two full-bandwidth links (L1 and L2) provide the following service:

- FC1: Pass full bandwidth data over full connection

The two low-bandwidth backup links (I1 and I2) provide the following service:

- DC1: Pass limited bandwidth data over degraded connection

The full-bandwidth backup link (I3) provides the following service:

- AC1: Pass full bandwidth data over alternate connection

A.3 Finite-State Machine Specification

S0: Initial state

N1: MCB1, MCB2

N2: BB1, BB2, BB3

N3: BB1, BB2, BB3

L1: FC1

L2: FC1

The first level of transitions from the initial state consist of a single fault occurring from the initial state. There are eight such transitions leading to the following eight states: S1, S2, S3, S4, S5, S6, S7, S8. The services after the transition include any alternate services started as a result of application reconfiguration.

S1: Process failure - MCB1 (N1)

N1: MCB2

N2: BB1, BB2, BB3, BB4

N3: BB1, BB2, BB3, BB4

L1: FC1

L2: FC1

S2: Database failure - MCB2 (N1)

N1: MCB1, MCB3

N2: BB1, BB2, BB3

N3: BB1, BB2, BB3

L1: FC1

L2: FC1

S3: Full node failure - MCB1, MCB2 (N1)

N1: -

N2: BB1, BB2, BB3, BB7

- N3: BB1, BB2, BB3, BB7
- L1: FC1
- L2: FC1
- I3: AC1
- S4: Process failure - BB1 (either N2 or N3)
 - N1: MCB1, MCB2
 - N2: BB2, BB3
 - N3: BB1, BB2, BB3
 - L1: FC1
 - L2: FC1
- S5: Process failure - BB2 (either N2 or N3)
 - N1: MCB1, MCB2, MCB3
 - N2: BB1, BB3
 - N3: BB1, BB2, BB3
 - L1: FC1
 - L2: FC1
- S6: Database failure - BB3 (either N2 or N3)
 - N1: MCB1, MCB2, MCB3
 - N2: BB1, BB2, BB4
 - N3: BB1, BB2, BB3
 - L1: FC1
 - L2: FC1
- S7: Full node failure - BB1, BB2, BB3 (either N2 or N3)
 - N1: MCB1, MCB2, MCB3
 - N2: -
 - N3: BB1, BB2, BB3
 - L1: FC1
 - L2: FC1
- S8: Link failure - FC1 (either L1 or L2)
 - N1: MCB1, MCB2
 - N2: BB1, BB2, BB3, BB6
 - N3: BB1, BB2, BB3, BB6
 - L1: -
 - L2: FC1
 - I1: DC1

To handle a second sequential fault from the initial state, it is necessary to specify the list of faults that can occur from each of the above states. Then, each fault would necessitate another transition in the finite-state machine to a different state. The following are the second level of states in the finite-state machine and the fault that causes the transition to each state:

From State S1:

- S10: Database failure - MCB2 (N1)
- S11: Full node failure - MCB2 (N1)
- S12: Process failure - BB1 (either N2 or N3)
- S13: Process failure - BB2 (either N2 or N3)
- S14: Database failure - BB3 (either N2 or N3)
- S15: Process failure - BB4 (either N2 or N3)
- S16: Full node failure - BB1, BB2, BB3, BB4 (either N2 or N3)
- S17: Link failure - FC1 (either L1 or L2)

From State S2:

- S20: Database failure - MCB2 (N1)
- S21: Process failure - MCB3 (N1)
- S22: Full node failure - MCB2, MCB3 (N1)
- S23: Process failure - BB1 (either N2 or N3)
- S24: Process failure - BB2 (either N2 or N3)
- S25: Database failure - BB3 (either N2 or N3)
- S26: Full node failure - BB1, BB2, BB3 (either N2 or N3)
- S27: Link failure - FC1 (either L1 or L2)

From State S3:

- S30: Process failure - BB1 (either N2 or N3)
- S31: Process failure - BB2 (either N2 or N3)
- S32: Database failure - BB3 (either N2 or N3)
- S33: Process failure - BB7 (either N2 or N3)
- S34: Full node failure - BB1, BB2, BB3, BB7 (either N2 or N3)
- S35: Link failure - FC1 (either L1 or L2)
- S36: Link failure - AC1 (I3)

From State S4:

- S40: Process failure - MCB1 (N1)
- S41: Database failure - MCB2 (N1)
- S42: Full node failure - MCB1, MCB2 (N1)
- S43: Process failure - BB2 (same N2 or N3 as previous fault)
- S44: Database failure - BB3 (same N2 or N3 as previous fault)
- S45: Full node failure - BB2, BB3 (same N2 or N3 as previous fault)
- S46: Process failure - BB1 (different N2 or N3 from previous fault)
- S47: Process failure - BB2 (different N2 or N3 from previous fault)
- S48: Database failure - BB3 (different N2 or N3 from previous fault)
- S49: Full node failure - BB1, BB2, BB3 (different N2 or N3 from previous fault)
- S4a: Link failure - FC1 (either L1 or L2)
- S4b: Link failure - AC1 (I3)

From State S5:

- S50: Process failure - MCB1 (N1)
- S51: Database failure - MCB2 (N1)
- S52: Process failure - MCB3 (N1)
- S53: Full node failure - MCB1, MCB2, MCB3 (N1)
- S54: Process failure - BB1 (same N2 or N3 as previous fault)
- S55: Database failure - BB3 (same N2 or N3 as previous fault)
- S56: Full node failure - BB1, BB3 (same N2 or N3 as previous fault)
- S57: Process failure - BB1 (different N2 or N3 from previous fault)
- S58: Process failure - BB2 (different N2 or N3 from previous fault)
- S59: Database failure - BB3 (different N2 or N3 from previous fault)
- S5a: Full node failure - BB1, BB2, BB3 (different N2 or N3 from previous fault)
- S5b: Link failure - FC1 (either L1 or L2)

From State S6:

- S60: Process failure - MCB1 (N1)
- S61: Database failure - MCB2 (N1)
- S62: Process failure - MCB3 (N1)
- S63: Full node failure - MCB1, MCB2, MCB3 (N1)

S64: Process failure - BB1 (same N2 or N3 as previous fault)
 S65: Database failure - BB2 (same N2 or N3 as previous fault)
 S66: Process failure - BB4 (same N2 or N3 as previous fault)
 S67: Full node failure - BB1, BB2, BB4 (same N2 or N3 as previous fault)
 S68: Process failure - BB1 (different N2 or N3 from previous fault)
 S69: Process failure - BB2 (different N2 or N3 from previous fault)
 S6a: Database failure - BB3 (different N2 or N3 from previous fault)
 S6b: Full node failure - BB1, BB2, BB3 (different N2 or N3 from previous fault)
 S6c: Link failure - FC1 (either L1 or L2)

From State S7:

S70: Process failure - MCB1 (N1)
 S71: Database failure - MCB2 (N1)
 S72: Process failure - MCB3 (N1)
 S73: Full node failure - MCB1, MCB2, MCB3 (N1)
 S74: Process failure - BB1 (different N2 or N3 from previous fault)
 S75: Process failure - BB2 (different N2 or N3 from previous fault)
 S76: Database failure - BB3 (different N2 or N3 from previous fault)
 S77: Full node failure - BB1, BB2, BB3 (different N2 or N3 from previous fault)
 S78: Link failure - FC1 (either L1 or L2)

From State S8:

S80: Process failure - MCB1 (N1)
 S81: Database failure - MCB2 (N1)
 S82: Full node failure - MCB1, MCB2 (N1)
 S83: Process failure - BB1 (either N2 or N3)
 S84: Process failure - BB2 (either N2 or N3)
 S85: Database failure - BB3 (either N2 or N3)
 S86: Process failure - BB6 (either N2 or N3)
 S87: Full node failure - BB1, BB2, BB3, BB6 (either N2 or N3)
 S88: Link failure - DC1 (same I1 or I2 that was started because of previous fault)
 S89: Link failure - FC1 (different L1 or L2 from previous fault)

A.4 Finite-State Machine Transitions

The transitions from the initial state to the next state caused by a single fault are described in this subsection. In the state descriptions above, the surviving services and the alternate services that were started are already specified. These actions describe the handling of the fault both in response to its occurrence and after the repair of that fault (to return to the initial state):

- Transition S0 to S1:
 Response: Start BB4 (both N2 and N3)
 Repair: Start BB5 (both N2 and N3)
 Stop BB4 (both N2 and N3)
 Stop BB5 (when each queue empty)
- Transition S0 to S2:
 Response: Start MCB3
 Repair: Start MCB4
 Stop MCB3
 Stop MCB4

- Transition S0 to S3:
Response: Start AC1
Start BB7 (both N2 and N3)
Repair: Stop BB7 (both N2 and N3)
Stop AC1
- Transition S0 to S4:
(No response)
- Transition S0 to S5:
Response: Start MCB3
Repair: Start MCB4
Stop MCB3
Stop MCB4 (when queue empty)
- Transition S0 to S6:
Response: Start MCB3
Start BB4 (for node N2 or N3 with fault)
Repair: Start MCB4
Stop MCB3
Stop MCB4 (when queue empty)
Stop MCB (for repaired node N2 or N3)
- Transition S0 to S7:
Response: Start MCB3
Repair: Start MCB4
Stop MCB3
Stop MCB4 (when queue empty)
- Transition S0 to S8:
Response: Start DC1 (for failed link L1 or L2)
Start BB6
Repair: Start BB5
Stop BB6
Stop BB5 (when queue empty)
Stop DC1 (for repaired link L1 or L2)

Appendix B

YACC Grammar for the RAPTOR Specification Notation

The following is the YACC grammar for the first-generation RAPTOR Specification Notation.

```
%%

%token SAS
%token SPMS
%token SIS
%token EDS
%token ERS
%token NODE
%token <stringtype> NODE_NAME
%token TYPE
%token <stringtype> TYPE_NAME
%token PROP
%token <stringtype> PROP_NAME
%token EVENT
%token <stringtype> EVENT_NAME
%token SERVICE
%token <stringtype> SERVICE_NAME
%token SET
%token <stringtype> SET_NAME
%token <stringtype> SET_MEMBER
%token <stringtype> VAR
%token <stringtype> COMPONENT_TYPE
%token ERROR
%token <stringtype> ERROR_NAME
%token FAILURE
%token ARROW
%token FORALL
%token EXISTS
%token AND
%token OR
%token IN
%token <inttype> START
%token <inttype> STOP

%type <inttype> critical_service

%%

raptor_specification
: sas spms sis eds ers
  { printf("Parsed a RAPTOR specification!!!\n"); }

sas
: SAS declaration_list propositions

declaration_list
: node_declarations
  { printf("CompletedNodes\n"); nodes.CompletedNodes(); }
| declaration_list type_declarations
  { printf("CompletedTypes\n"); nodes.CompletedTypes(); }
| declaration_list prop_declarations
```

```

        { printf("CompletedProps\n"); nodes.CompletedProps(); }

node_declarations
    : node_declaration
    | node_declarations node_declaration

node_declaration
    : NODE node_list

node_list
    : NODE_NAME
    { nodes.AddNode($1); }
    | node_list ',' NODE_NAME
    { nodes.AddNode($3); }

type_declarations
    : type_declaration
    | type_declarations type_declaration

type_declaration
    : TYPE TYPE_NAME
    { nodes.AddType($2); }

prop_declarations
    : prop_declaration
    | prop_declarations prop_declaration

prop_declaration
    : PROP PROP_NAME
    { nodes.AddProp($2); }

propositions
    : proposition
    | propositions proposition

proposition
    : TYPE_NAME '(' NODE_NAME ')' ';'
    { nodes.SetNodeType($3, $1); }
    | PROP_NAME '(' NODE_NAME ')' ';'
    { nodes.SetNodeProp($3, $1); }

spms
    : SPMS service_mappings
    { printf("CompletedMappings\n");
      nodes.CompletedNodeServiceMapping(); }

service_mappings
    : service_mapping
    | service_mappings service_mapping

service_mapping
    : single_node_service_mapping
    | multiple_node_service_mapping

single_node_service_mapping
    : NODE_NAME ARROW service_list ';'
    { printf("Mapping services to node\n");

```

```

        nodes.MapServicesToNode($1); }

multiple_node_service_mapping
: FORALL TYPE_NAME ARROW service_list ';'
  { printf("Mapping services to type\n");
    nodes.MapServicesToType($2); }

service_list
: SERVICE_NAME
  { nodes.AddService($1); }
| service_list ',' SERVICE_NAME
  { nodes.AddService($3); }

sis
: SIS set_declaration_list set_definitions

set_declaration_list
: set_declarations
  { printf("CompletedSets\n");
    nodes.CompletedSets(); }

set_declarations
: set_declaration
| set_declarations set_declaration

set_declaration
: SET SET_NAME
  { nodes.AddSet($2); }

set_definitions
: set_definition
| set_definitions set_definition

set_definition
: SET_NAME '=' '{' set_list '}'
  { nodes.SetSet($1); }
| SET_NAME '=' '{' set_iteration '}'
  { nodes.SetSet($1); }

set_list
: SET_MEMBER
  { nodes.AddToTempSetList($1); }
| set_list ',' SET_MEMBER
  { nodes.AddToTempSetList($3); }

set_iteration
: VAR ':' COMPONENT_TYPE '|' PROP_NAME '(' VAR ')'
  { //printf("Vars = %s|s, Type = %s, Prop = %s\n", $1, $7, $3, $5);
    nodes.AddPropToTempSet($5); }
| VAR ':' COMPONENT_TYPE '|' TYPE_NAME '(' VAR ')'
  { //printf("Vars = %s|s, Type = %s, Prop = %s\n", $1, $7, $3, $5);
    nodes.AddTypeToTempSet($5); }

eds
: EDS error_declaration_list error_definitions

error_declaration_list

```

```

        : error_declarations
          { printf("CompletedErrors\n");
            nodes.CompletedErrors(); }

error_declarations
  : error_declaration
  | error_declarations error_declaration

error_declaration
  : ERROR ERROR_NAME
    { nodes.AddError($2); }

error_definitions
  : error_definition
  | error_definitions error_definition

error_definition
  : ERROR_NAME '=' conditions ';'

conditions
  : condition
  | '(' condition ')'
  | conditions conjunction condition
  | '(' conditions conjunction condition ')'

condition
  : FAILURE '(' NODE_NAME '.' SERVICE_NAME ')'
    { printf("Failure: Node = %s, Serv = %s\n", $3, $5); }
  | EVENT_NAME '(' NODE_NAME ')'
    { printf("Event = %s, Node = %s\n", $1, $3); }
  | '(' FORALL VAR IN SET_NAME '|' EVENT_NAME '(' VAR ')' ')'
    { printf("Var = %s|%s, Set = %s, Event = %s\n", $3, $9, $5, $7); }
  | '(' EXISTS VAR IN SET_NAME '|' EVENT_NAME '(' VAR ')' ')'
    { printf("Var = %s|%s, Set = %s, Event = %s\n", $3, $9, $5, $7); }

conjunction
  : AND
  | OR

ers
  : ERS error_activities
    { printf("CompletedResponses\n"); }

error_activities
  : per_error_activities
  | error_activities per_error_activities

per_error_activities
  : ERROR_NAME ':' per_node_activities
    { nodes.CompletedResponse($1); }

per_node_activities
  : per_node_responses
  | per_node_activities per_node_responses

per_node_responses
  : NODE_NAME ARROW response_list ';'

```

```

    { nodes.CompletedNodeResponse($1); }
| SET_NAME ARROW response_list ';'
    { nodes.CompletedSetResponse($1); }

response_list
: SERVICE_NAME '.' critical_service
  { nodes.AddResponse($1, $3); }
| response_list ',' SERVICE_NAME '.' critical_service
  { nodes.AddResponse($3, $5); }

critical_service
: START
  { $$ = CS_START; }
| STOP
  { $$ = CS_STOP; }

%%

```

Appendix C

100-Node Banking Example

The following is the RAPTOR specification of error recovery for a 100-node banking system.

SYSTEM_ARCHITECTURE_SPECIFICATION

```

NODE frb1, frb2, frb3
NODE mcb100, mcb200, mcb300, mcb400, mcb500, mcb600, mcb700, mcb800, mcb900,
    mcb1000
NODE bb101, bb102, bb103, bb104, bb105, bb106, bb107, bb108, bb109
NODE bb201, bb202, bb203, bb204, bb205, bb206, bb207, bb208, bb209
NODE bb301, bb302, bb303, bb304, bb305, bb306, bb307, bb308, bb309
NODE bb401, bb402, bb403, bb404, bb405, bb406, bb407, bb408, bb409
NODE bb501, bb502, bb503, bb504, bb505, bb506, bb507, bb508, bb509
NODE bb601, bb602, bb603, bb604, bb605, bb606, bb607, bb608, bb609
NODE bb701, bb702, bb703, bb704, bb705, bb706, bb707, bb708, bb709
NODE bb801, bb802, bb803, bb804, bb805, bb806, bb807, bb808, bb809
NODE bb901, bb902, bb903, bb904, bb905, bb906, bb907, bb908, bb909
NODE bb1001, bb1002, bb1003, bb1004, bb1005, bb1006, bb1007, bb1008, bb1009

TYPE federal_reserve
TYPE money_center
TYPE branch

PROP east_coast
PROP north_east
PROP south_east
PROP north_central
PROP south_central
PROP north_west
PROP south_west
PROP west_coast

EVENT security_attack
EVENT node_failure
EVENT power_failure

federal_reserve(frb1); federal_reserve(frb2); federal_reserve(frb3);
money_center(mcb100);
branch(bb101); branch(bb102); branch(bb103);
branch(bb104); branch(bb105); branch(bb106);
branch(bb107); branch(bb108); branch(bb109);
money_center(mcb200);
branch(bb201); branch(bb202); branch(bb203);
branch(bb204); branch(bb205); branch(bb206);
branch(bb207); branch(bb208); branch(bb209);
money_center(mcb300);
branch(bb301); branch(bb302); branch(bb303);
branch(bb304); branch(bb305); branch(bb306);
branch(bb307); branch(bb308); branch(bb309);
money_center(mcb400);
branch(bb401); branch(bb402); branch(bb403);
branch(bb404); branch(bb405); branch(bb406);
branch(bb407); branch(bb408); branch(bb409);
```

```

money_center(mcb500);
branch(bb501); branch(bb502); branch(bb503);
branch(bb504); branch(bb505); branch(bb506);
branch(bb507); branch(bb508); branch(bb509);
money_center(mcb600);
branch(bb601); branch(bb602); branch(bb603);
branch(bb604); branch(bb605); branch(bb606);
branch(bb607); branch(bb608); branch(bb609);
money_center(mcb700);
branch(bb701); branch(bb702); branch(bb703);
branch(bb704); branch(bb705); branch(bb706);
branch(bb707); branch(bb708); branch(bb709);
money_center(mcb800);
branch(bb801); branch(bb802); branch(bb803);
branch(bb804); branch(bb805); branch(bb806);
branch(bb807); branch(bb808); branch(bb809);
money_center(mcb900);
branch(bb901); branch(bb902); branch(bb903);
branch(bb904); branch(bb905); branch(bb906);
branch(bb907); branch(bb908); branch(bb909);
money_center(mcb1000);
branch(bb1001); branch(bb1002); branch(bb1003);
branch(bb1004); branch(bb1005); branch(bb1006);
branch(bb1007); branch(bb1008); branch(bb1009);

east_coast(frb1); south_east(frb2); south_central(frb3);
east_coast(mcb100);
east_coast(bb101); east_coast(bb102); north_east(bb103);
south_east(bb104); north_central(bb105); south_central(bb106);
north_west(bb107); south_west(bb108); west_coast(bb109);
east_coast(mcb200);
east_coast(bb201); east_coast(bb202); north_east(bb203);
south_east(bb204); north_central(bb205); south_central(bb206);
north_west(bb207); south_west(bb208); west_coast(bb209);
north_east(mcb300);
east_coast(bb301); north_east(bb302); north_east(bb303);
south_east(bb304); north_central(bb305); south_central(bb306);
north_west(bb307); south_west(bb308); west_coast(bb309);
south_east(mcb400);
east_coast(bb401); north_east(bb402); south_east(bb403);
south_east(bb404); north_central(bb405); south_central(bb406);
north_west(bb407); south_west(bb408); west_coast(bb409);
north_central(mcb500);
east_coast(bb501); north_east(bb502); south_east(bb503);
north_central(bb504); north_central(bb505); south_central(bb506);
north_west(bb507); south_west(bb508); west_coast(bb509);
south_central(mcb600);
east_coast(bb601); north_east(bb602); south_east(bb603);
north_central(bb604); south_central(bb605); south_central(bb606);
north_west(bb607); south_west(bb608); west_coast(bb609);
south_central(mcb700);
east_coast(bb701); north_east(bb702); south_east(bb703);
north_central(bb704); south_central(bb705); south_central(bb706);
north_west(bb707); south_west(bb708); west_coast(bb709);
north_west(mcb800);
east_coast(bb801); north_east(bb802); south_east(bb803);
north_central(bb804); south_central(bb805); north_west(bb806);

```

```

north_west(bb807); south_west(bb808); west_coast(bb809);
south_west(mcb900);
east_coast(bb901); north_east(bb902); south_east(bb903);
north_central(bb904); south_central(bb905); north_west(bb906);
south_west(bb907); south_west(bb908); west_coast(bb909);
west_coast(mcb1000);
east_coast(bb1001); north_east(bb1002); south_east(bb1003);
north_central(bb1004); south_central(bb1005); north_west(bb1006);
south_west(bb1007); west_coast(bb1008); west_coast(bb1009);

```

SERVICE_PLATFORM_MAPPING_SPECIFICATION

```

FORALL federal_reserve -> route_batch_requests, route_batch_responses,
                           db_mc_balances,
                           frb_actuator_alert_on,
                           frb_actuator_alert_off,
                           frb_actuator_primary_frb_assignment,
                           frb_actuator_system_shutdown;

FORALL money_center -> route_requests, route_responses,
                       db_branch_balances,
                       batch_requests, send_batch_requests,
                       process_batch_requests,
                       send_batch_responses, process_batch_responses,
                       mcb_actuator_alert_on,
                       mcb_actuator_alert_off,
                       mcb_actuator_new_primary_frb,
                       mcb_actuator_system_shutdown;

FORALL branch -> accept_requests, send_requests_up,
                 db_account_balances,
                 receive_requests, process_requests,
                 send_responses_up,
                 process_responses, send_responses_down,
                 bb_actuator_system_shutdown;

```

SYSTEM_INTERFACE_SPECIFICATION

```

SET FederalReserveBanks
SET MoneyCenterBanks
SET BranchBanks
SET PrimaryFederalReserve
SET FederalReserveBackups
SET EastCoastBanks
SET NorthEastBanks
SET SouthEastBanks
SET NorthCentralBanks
SET SouthCentralBanks
SET NorthWestBanks
SET SouthWestBanks
SET WestCoastBanks
SET CitibankBanks
SET ChaseManhattanBanks

```

```

FederalReserveBanks = { frb1, frb2, frb3 }
MoneyCenterBanks = { i : NODE | money_center(i) }
BranchBanks = { i : NODE | branch(i) }
PrimaryFederalReserve = { frb1 }
FederalReserveBackups = { frb2, frb3 }
EastCoastBanks = { i : NODE | east_coast(i) }
NorthEastBanks = { i : NODE | north_east(i) }
SouthEastBanks = { i : NODE | south_east(i) }
NorthCentralBanks = { i : NODE | north_central(i) }
SouthCentralBanks = { i : NODE | south_central(i) }
NorthWestBanks = { i : NODE | north_west(i) }
SouthWestBanks = { i : NODE | south_west(i) }
WestCoastBanks = { i : NODE | west_coast(i) }
CitibankBanks = {mcb100, bb101, bb102, bb103, bb104, bb105, bb106, bb107,
bb108, bb109}
ChaseManhattanBanks = {mcb200, bb201, bb202, bb203, bb204, bb205, bb206,
bb207, bb208, bb209}

```

ERROR_DETECTION_SPECIFICATION

```

ERROR PrimaryFrbFailure
ERROR McbSecurityAttack
ERROR CoordinatedAttack
ERROR WidespreadPowerFailure

```

```

PrimaryFrbFailure =
    (EXISTS i IN PrimaryFederalReserve | node_failure(i) OR power_failure(i));

```

```

McbSecurityAttack =
    (EXISTS i IN MoneyCenterBanks | security_attack(i));

```

```

CoordinatedAttack =
    ( (EXISTS i IN FederalReserveBanks | security_attack(i))
      AND
      (EXISTS i IN MoneyCenterBanks | security_attack(i)))
    OR
    ( FORALL i IN MoneyCenterBanks | security_attack(i));

```

```

WidespreadPowerFailure =
    ( FORALL i IN EastCoastBanks | power_failure(i))
    OR
    ( FORALL i IN NorthEastBanks | power_failure(i))
    OR
    ( FORALL i IN SouthEastBanks | power_failure(i))
    OR
    ( FORALL i IN NorthCentralBanks | power_failure(i))
    OR
    ( FORALL i IN SouthCentralBanks | power_failure(i))
    OR
    ( FORALL i IN NorthWestBanks | power_failure(i))
    OR
    ( FORALL i IN SouthWestBanks | power_failure(i))
    OR
    ( FORALL i IN WestCoastBanks | power_failure(i));

```

ERROR_RECOVERY_SPECIFICATION

```
PrimaryFrbFailure(NODE): action_1
    PrimaryFrbFailure(NODE): action_1_1
        PrimaryFrbFailure(NODE): action_1_1_1
        McbSecurityAttack(NODE): action_1_1_2
        CoordinatedAttack(): action_1_1_3
        WidespreadPowerFailure(SET): action_1_1_4
    McbSecurityAttack(NODE): action_1_2
        PrimaryFrbFailure(NODE): action_1_2_1
        McbSecurityAttack(NODE): action_1_2_2
        CoordinatedAttack(): action_1_2_3
        WidespreadPowerFailure(SET): action_1_2_4
    CoordinatedAttack(): action_1_3
    WidespreadPowerFailure(SET): action_1_4
        PrimaryFrbFailure(NODE): action_1_4_1
        McbSecurityAttack(NODE): action_1_4_2
        CoordinatedAttack(): action_1_4_3
        WidespreadPowerFailure(SET): action_1_4_4

McbSecurityAttack(NODE): action_2
    PrimaryFrbFailure(NODE): action_2_1
        PrimaryFrbFailure(NODE): action_2_1_1
        McbSecurityAttack(NODE): action_2_1_2
        CoordinatedAttack(): action_2_1_3
        WidespreadPowerFailure(SET): action_2_1_4
    McbSecurityAttack(NODE): action_2_2
        PrimaryFrbFailure(NODE): action_2_2_1
        McbSecurityAttack(NODE): action_2_2_2
        CoordinatedAttack(): action_2_2_3
        WidespreadPowerFailure(SET): action_2_2_4
    CoordinatedAttack(): action_2_3
    WidespreadPowerFailure(SET): action_2_4
        PrimaryFrbFailure(NODE): action_2_4_1
        McbSecurityAttack(NODE): action_2_4_2
        CoordinatedAttack(): action_2_4_3
        WidespreadPowerFailure(SET): action_2_4_4

CoordinatedAttack(): action_3

WidespreadPowerFailure(SET): action_4
    PrimaryFrbFailure(NODE): action_4_1
        PrimaryFrbFailure(NODE): action_4_1_1
        McbSecurityAttack(NODE): action_4_1_2
        CoordinatedAttack(): action_4_1_3
        WidespreadPowerFailure(SET): action_4_1_4
    McbSecurityAttack(NODE): action_4_2
        PrimaryFrbFailure(NODE): action_4_2_1
        McbSecurityAttack(NODE): action_4_2_2
        CoordinatedAttack(): action_4_2_3
        WidespreadPowerFailure(SET): action_4_2_4
    CoordinatedAttack(): action_4_3
    WidespreadPowerFailure(SET): action_4_4
        PrimaryFrbFailure(NODE): action_4_4_1
```

```

McbSecurityAttack(NODE): action_4_4_2
CoordinatedAttack(): action_4_4_3
WidespreadPowerFailure(SET): action_4_4_4

```

```

action_1(NODE frb_num):
action_1_1(NODE frb_num):
action_1_2_1(NODE frb_num):
action_1_4_1(NODE frb_num):
action_2_1(NODE frb_num):
action_2_1_1(NODE frb_num):
action_2_2_1(NODE frb_num):
action_2_4_1(NODE frb_num):
action_4_1(NODE frb_num):
action_4_1_1(NODE frb_num):
action_4_2_1(NODE frb_num):
action_4_4_1(NODE frb_num):
    frb_num -> shutdown();
    REMOVE(FederalReserveBanks, frb_num);
    REMOVE(PrimaryFederalReserve, frb_num);
    FederalReserveBanks -> reconfig_frb_down(frb_num);

```

```

action_1_1_2(NODE mcb_num):
action_1_2(NODE mcb_num):
action_1_2_2(NODE mcb_num):
action_1_4_2(NODE mcb_num):
action_2(NODE mcb_num):
action_2_1_2(NODE mcb_num):
action_2_2_2(NODE mcb_num):
action_2_4_2(NODE mcb_num):
action_4_1_2(NODE mcb_num):
action_4_2(NODE mcb_num):
action_4_2_2(NODE mcb_num):
action_4_4_2(NODE mcb_num):
    FederalReserveBanks -> raise_alert();
    mcb_num -> reconfig_mcb_attacked(mcb_num);

```

```

action_1_1_1(NODE frb_num):
action_1_1_3():
action_1_2_3():
action_1_3():
action_1_4_3():
action_2_1_3():
action_2_2_2(NODE mcb_num):
action_2_2_3():
action_2_3():
action_2_4_3():
action_3():
action_4_1_3():
action_4_2_3():
action_4_3():
action_4_4_3():
    BranchBanks -> shutdown();
    MoneyCenterBanks -> shutdown();
    FederalReserveBanks -> shutdown();

```

```

action_1_1_4(SET region):

```

```

action_1_2_4(SET region):
action_1_4(SET region):
action_1_4_4(SET region):
action_2_1_4(SET region):
action_2_2_4(SET region):
action_2_4(SET region):
action_2_4_4(SET region):
action_4(SET region):
action_4_1_4(SET region):
action_4_2_4(SET region):
action_4_4(SET region):
action_4_4_4(SET region):
    // if primary frb in region, promote another FRB
    // if any mcbs in region, promote BBs in other region
    switch(region)
        case east_coast:
            frb1 -> shutdown();
            REMOVE(FederalReserveBanks, frb1);
            FederalReserveBanks -> reconfig_frb_down(frb1);
            mcb100 -> shutdown();
            REMOVE(MoneyCenterBanks, mcb100);
            bb103 -> promote_to_mcb();
            ADD(MoneyCenterBanks, bb103);
            CitibankBanks -> reconfig_mcb_down(mcb100, bb103);
            mcb200 -> shutdown();
            REMOVE(MoneyCenterBanks, mcb200);
            bb203 -> promote_to_mcb();
            ADD(MoneyCenterBanks, bb203);
            ChaseManhattanBanks -> reconfig_mcb_down(mcb200, bb203);

```

APPENDIX B

“A Security Architecture for Survivable Systems”

A SECURITY ARCHITECTURE FOR SURVIVABLE SYSTEMS

Chenxi Wang, John C. Knight

*Department of Computer Science
University of Virginia*

*151 Engineer's Way, P.O. Box 400740
Charlottesville, VA 22904-4740*

cw2et@cs.virginia.edu

knight@cs.virginia.edu

(804) 982-2216

Abstract

The protection of survivability mechanisms against security attacks is a difficult but extremely important problem. If this mechanism were penetrated in any particular system, the adversary might gain control of the entire associated information system. What is needed is a mechanism to preserve the execution integrity of software despite its untrustworthy execution environment. In this paper, we present a comprehensive strategy for protecting survivability mechanisms against attack by adversaries with access to significant resources. The approach uses a variety of forms of diversity at the system level and a general strategy for defeating static analysis at the local level. We refer to the approach to defeating static analysis as One-Way translation and describe the concepts, the underlying theory, the performance, and the implementation.

A SECURITY ARCHITECTURE FOR SURVIVABLE SYSTEMS

1 INTRODUCTION

Today's society is heavily dependent on a number of large information systems that constitute the foundation of everyday activities. The management of transportation systems such as freight rail and air traffic control, the operation of telecommunications systems, nationwide control of electric power generation and distribution, and the basic operation of the financial system are but a few examples of this class of information system. Such systems are often referred to as *critical infrastructure systems*.

Societal dependence on these systems is growing and will continue to grow for the foreseeable future. This has raised concerns about the dependability and survivability of these infrastructure systems [PCCI97, DSB96]. Recent experience has shown that these systems fail for many reasons including hardware failure, software failure, operator errors, and malicious attacks [KEP97, RISK94].

Researchers have proposed a number of approaches intended to improve the survivability of critical infrastructure systems, many of which employ network-wide architectural support to enhance system survivability. These schemes often rely on a distributed monitoring mechanism in which the system in its entirety is monitored in order to detect undesirable state changes including benign faults, malicious behaviors, and other anomalies [PONE97, GRID97, IDIP97, NEUM00]. A number of systems also deploy automated response mechanisms in which real-time changes to the system are determined based on the monitoring information, and the system is reconfigured to reflect the changes ([IDIP97, SKDG98]).

Unfortunately, the introduction of network-wide monitoring-and-control elements can lead to new vulnerabilities. For example, if the control elements were to fail, the failure might result in a system-wide loss of services. Far more significant, however, is the fact that these elements can themselves be the target of malicious attacks. If an intruder were able to corrupt the monitors, he or she could send fraudulent state information to the controllers to cause erroneous state changes. Moreover, if the control elements are penetrated, the intruder would gain access to the control of the entire network simply by manipulating the control mechanism. This is in contrast to a typical network intrusion in which an attacker might be able to compromise a single node (albeit perhaps an important one) but where access beyond there usually requires additional knowledge and resources.

Protection of the network-wide survivability elements is, therefore, essential to survivable systems. In part, traditional security mechanisms can be employed. For example, some components of the control mechanism can be executed on dedicated hosts that are physically isolated and protected, and cryptographic techniques could be used to prevent tampering of communications between these control hosts and the rest of the system. However, parts of the control mechanism need to reside on the monitored hosts to collect data and perform local reconfiguration tasks, and

it is the protection of these components that is the most difficult: it must be assumed that the monitored hosts are vulnerable to security attacks (hence, in part, the need for security monitoring). We must allow the possibility that any software running on top of the monitored hosts can be compromised, and that some adversary will try to impersonate or tamper with the program to perform malicious tasks.

What is needed here is a mechanism to preserve the execution integrity of software despite its untrustworthy execution environment. This is exactly the motivation for this research. This document contains a comprehensive, high-level description of the problem and the devised solution to date. As such, this paper should be utilized mainly as a documentation of creative thoughts and a roadmap of the research.

The rest of the document is organized as follows: Section 2 presents the problem context, assumptions and scope of this research. Section 3 describes the fundamental observations that constitute the underlying principles of our approach. Section 4 presents the solution framework and a few examples of the techniques. Section 5 analyzes the security strength of the approach. Section 6 presents related work, and Section 7 concludes the paper by presenting future research directions.

2 THE SYSTEM CONTEXT AND ASSUMPTIONS

In this section, we describe the system context and the assumptions based on which this research is established. We first review the relevant characteristics of critical infrastructure systems and the survivability architecture, as these characteristics provide the necessary backdrop for the solution approach. We then present a set of assumptions to set the problem context.

2.1 Critical Infrastructure Systems

Critical infrastructure systems are complex by nature. Listed below are some characteristics of infrastructure systems that are important to this research.

- **Large scale:** Critical infrastructure systems often involve a large number of computing nodes that are geographically dispersed. These computing nodes and their interconnections provide the computation, data storage, and communication that are needed to provide services. The exact topology of the network is not important, although such networks are usually not fully connected. A typical infrastructure system uses point-to-point links between computing nodes, and the nodes can communicate with one another in any fashion that is desired by the applications.
- **Heterogeneity:** Another defining characteristic of critical infrastructure systems is the degree of heterogeneity. First, the infrastructure system is often composed of subsystems with diverse operational policies and environments—consider regional banks within the same banking infrastructure—and a wide variety of software and hardware components. From a security standpoint, this degree of heterogeneity implies incongruity in the security policies and mechanisms employed; some sites will be more easily penetrated than others. A universal protection mechanism such as securing each individual host from the ground up is both impractical and infeasible.

Second, critical applications usually do not consist of a set of similar programs (such as mail

servers) cooperating to achieve some goal. Rather, the programs running on different hosts often serve distinct purposes, and they must cooperate in some form of sequential processing in order to provide desired services. A direct consequence of this diversity is that functionality is not uniformly distributed across the system; some computing nodes provide services that are more important than the services provided by others. In order to make system-level management decisions, it is therefore necessary to correlate and integrate local information. This implies complexity in enforcing survivability as well as inherent difficulties in securing the survivability mechanism.

- **Legacy and COTS components:** The software employed in infrastructure systems tends to be large and to make extensive use of both legacy and Commercial-Off-The-Shelf (COTS) components. This means that any mechanism we introduce must consider the uncertainty of COTS and legacy software in terms of their reliability and security. Moreover, the characteristics of the operational environment are determined largely by the applications—retrofitting the system with survivability mechanisms is particularly difficult for one cannot mandate drastic changes to existing system architectures (such as demanding changes to the network topology) or system and application software (such as demanding that the applications be rewritten).

2.2 The Survivability Architecture

It is in the context just described that survivability mechanisms have been proposed. An example mechanism is the hierarchic, adaptive, control-system architecture developed here at the University of Virginia [SKDG98]. The research henceforth is carried out in the context of this architecture. We briefly describe this survivability architecture below.

2.2.1 A Control System Construct

A critical component in the survivability architecture is the *control system* construct. Introduced as an external entity to manage the infrastructure system, the control system operates in parallel with the infrastructure system. Its primary function is to monitor the system operation, and make management decisions to enforce system survivability. For example, the control system may initiate dynamic system reconfigurations in order to continue services in the event of failures or security attacks.

The control system consists of a set of control servers and a collection of probing and actuating programs that execute on infrastructure hosts. The probe programs collect information about the critical application and send that to the control servers to be analyzed. The actuators accept reconfiguration commands from the control servers and initiate local changes that are implied by the control analysis. An abstracted view of such a control system architecture is shown in Figure 1.

The control hosts (represented as gray circles) are organized in a hierarchical structure with control actions being carried out in a number of different levels. They interact with the controlled system through a set of sensors and actuators (represented as black dots), which execute on the infrastructure hosts (represented as white circles). The sensors and actuators communicate monitoring information to the control servers and perform reconfiguration commands issued from the control servers. Control hosts are interconnected as appropriate.

Note the control servers are physically separate from the rest of the system. There are several advantages to such a design. First, executing control algorithms locally on application hosts can be a significant resource drain—running them exclusively on the control servers is beneficial for efficiency reasons. Second, there tend to be fewer numbers of control servers than there are application hosts, therefore it is possible to have dedicated control servers, and to secure and configure them independently. This implies a rigorously controlled execution environment for the control servers, and consequently, enhanced security and assurance.

2.2.2 The Probing and Actuating Mechanism

The primary function of the probes is to collect system state information as input and generate sensing values that are sent to the control servers for processing. Communications between the probes and the control servers follow a prescribed protocol that includes a timing mechanism (e.g. time-outs), predetermined data format, and designated hand-shaking sequences. Details of this communication will be discussed in later sections. It is important to note that the sensor program communicates with the control servers in a prescribed fashion, and adhering to the interaction protocol is considered expected behavior of the sensor program.

If necessary, the control servers make response decisions and communicate them back to the actuators to take effect. The actuator process accepts the control command and undertakes the actions needed locally to reconfigure the application. Examples of such dynamic reconfigurations include shutting down communication lines, dynamic process migration, active load balancing, etc. Interested readers can find details of this actuating process in other reports [ELKN99].

The probing and actuating functionality need to reside on the monitored host to take effect. They can be implemented as either separate processes or as distinct functionality within a single process. In the rest of this work, we will refer to the probe and actuator on the same host as a collective entity—the monitor process.

2.3 Security Issues

The survivability architecture, as described above, gives rise to a series of security concerns. In a

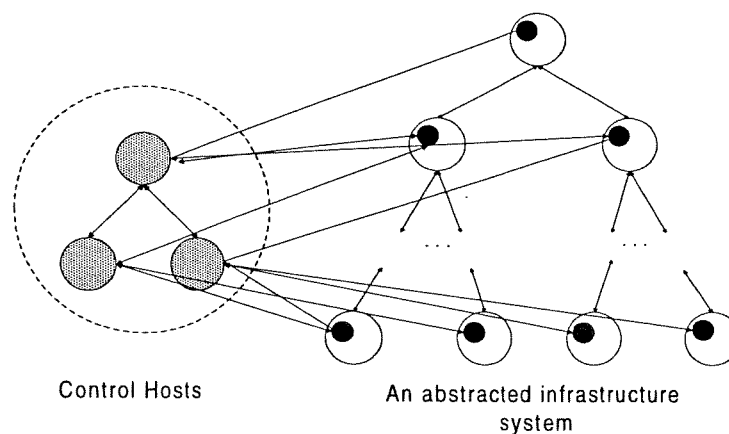


Figure 1. Example Controlled Infrastructure System

nutshell, these security concerns can be grouped into three loosely-defined categories.

- **Protection of the control servers.** The control servers execute the control algorithms, and therefore are at the heart of the whole control mechanism. The servers and the software executing on top of them must be protected from corruption and denial-of-service attacks.
- **Protection of communication.** Monitoring and control communications occur over the network. This network traffic needs to be authenticated, and protected against tampering, eavesdropping, insertion and deletion attacks.
- **Protection of the monitors.** The monitor process needs to be protected so that the data gathering and the actuating part of the mechanism can be trusted.

These protection issues are closely related; that is, techniques used to secure one part of the mechanism will impact the protection of others. For example, cryptographic methods are often used to protect network communication. However, cryptography is not sufficient if the communicating hosts (and consequently the secrecy of the cryptographic keys), are not adequately protected.

Because of the multitude of security issues, complete protection of the survivability architecture in its entirety, i.e., against every possible combination of attack scenarios, is beyond the immediate scope of this work. To simplify the problem and to help focus the task of this research work, we make the following assumptions:

- **Trustworthy control servers.** The control servers, physically separated from the rest of the system, are *dedicated* to performing the algorithmic part of the control mechanism. Securing the control servers requires a combination of careful system administration (e.g. exercise strict control on what software is allowed to execute on these servers) and good physical security (e.g. restricted access). For the purpose of this work, we assume that the control servers are secure and trustworthy; that is, the possibility of a successful security attack on the control servers is considered sufficiently low such that the software running on top of them can be trusted to perform its task correctly.
- **Secure communications.** We assume that all network communications occur over authenticated channels equipped with cryptographic protections, and that the cryptographic techniques are sufficient, in this context, to protect the freshness, integrity and privacy of the communication. It is important to note that the use of cryptography does not prevent denial-of-service attacks—network traffic can be deleted and communication channels can be jammed. An exception is the inter-server communication among control servers. The control servers communicate with one another over dedicated links with restricted access, and these links are assumed secure against eavesdrop, traffic insertion and deletion attacks.
- **Un-enhanced application hosts.** The infrastructure system is a network enterprise with tens of thousands of computing hosts. It is virtually impossible to enhance every application host to be rid of security concerns. We hence assume that application hosts are running in their normal operational mode—no special enhancement other than the security mechanisms that are already in place for the applications. This assumption has many implications. In particular, it implies that the application software and infrastructure hosts might contain security flaws, and that they might be vulnerable to a variety of security attacks.

- **Trusted survivability software.** Many real-world security problems arise not because of flaws or oversight in the design of the security mechanism, but rather they are caused by errors in the software implementation. It is not our objective, in this work, to address security flaws of the latter kind. Throughout this work, we assume that the survivability software is trusted in such a way that it operates as expected (if not compromised), and it does not contain any *malicious* flaws that will lead to a compromise of the survivability mechanism. This assumption allows us to focus on security threats from the environment and protection issues that are not implementation specific.

These assumptions state that, in the final analysis, the weak link lies in the protection of the probing and actuating mechanism on untrustworthy application hosts. Furthermore, we are primarily interested in sophisticated attacks that fall in the category of *intelligent tampering* and *impersonation* attacks:

- **Intelligent Tampering.** Intelligent tampering refers to scenarios in which an adversary modifies the program or data in some specific way that allows the program to continue to operate in a seemingly unaffected manner (from the trusted server's point of view), but on corrupted state or data. Overwriting data buffers with data of the correct format but different values is an example of such an attack. Under this definition, tampering with the software in a random way (e.g., overwriting random bits in the memory) does not constitute an intelligent tampering attack.
- **Impersonation.** An impersonation attack is accomplished when an intruder successfully emulates the observable behavior of the legitimate program. Similar to intelligent tampering, impersonation attacks seek to establish a rogue version of the legitimate program. The difference lies in that tampering deals with internal specifics of the program while impersonation operates at the level of observable semantics.

Both intelligent tampering and impersonation require some level of understanding about the target program—whether it is knowledge about the external behavior or the internal specifics of the program, some information is needed a priori for such an attack to succeed. Consider a scenario in which the intruder's objective is to forge monitor messages to the control server, and assume all messages are signed with the monitor's private key. The knowledge the intruder must acquire, among others, is the private key of the monitor program, for message forging is not possible without the correct signature key.

The ways in which an intruder can obtain the knowledge necessary to compromise the program depend heavily on the intruder's capabilities and resources. These capabilities and resources also have a significant impact on the design of the defense mechanism. Further assumptions about the intruder's capabilities and the attack scenarios have to be made before any analysis can be undertaken.

Three categories of intruders, classified by their respective capabilities, are likely to be present in the context of this work. Listed in the order of increasing level of capability, they are: *Network Intruders*, *Malicious Insiders*, and *Privileged Users*. This classification is based on the one used by Aucsmith [AUCS96] in his Integrity Verification Kernel (IVK) work. This section discusses each category within the context of the survivability architecture described earlier.

Network Intruders: This category refers to intruders who do not have direct access to the host where the monitor program executes. These intruders access the system through network entry points, and they can eavesdrop on the communication line, and insert and delete network traffic.

A typical network intruder is bounded by communications protocols and other network-based security mechanisms (e.g. firewalls, network access control, etc.). Their mission is to either breach the host security perimeter (i.e. getting in from outside) or become a *man-in-the-middle* by forging or replaying communication messages. This type of intrusion is best dealt with by the deployment of correctly designed and implemented security protocols and proper administration, and therefore is not covered by this work.

It should be noted that network intruders can gain further access by exploiting flaws in communication protocols or network security mechanisms. For example, a successful buffer overrun attack may render more privileges to the intruder as a result. In doing so, intruders from the network may become malicious insiders or privileged users who could possess significantly more powerful capabilities than a typical network intruder.

Malicious Insiders: This category refers to adversaries that have control of some program running on the targeted host. These intruders can be legitimate users of the system, or an outsider who has gained illegal access to the host system.

Malicious insiders have access to some system resource, and they can manipulate the programs under their control or introduce Trojan-horse programs to inflict damage to other applications or the underlying host system. An example of such a malicious insider is someone who has obtained the password of other users and is now able to read and write private data and programs that belong to the compromised accounts. In general, actions of malicious insiders do not directly undermine the security of the host system (e.g. they generally do not compromise the operating system). For the purpose of this discussion, It is assumed that these adversaries are still bounded by the operating system and its security mechanisms.

Malicious insiders are intruders without the “root privilege”. Actions of a malicious insider can be greatly limited by the use of properly designed access control mechanisms, competent intrusion detection tools and careful administration. At best, intruders in this category can cause denial of services or instantiate malicious software such as virus or Trojan-horse programs to effect damage to the target program. However, they usually do not employ sophisticated program analysis tools, and hence are not candidates for intelligent tampering and impersonation attacks. To defeat malicious insiders, the approach described here relies on the principle of diversity to reduce software uniformity, which is often the cause for successful virus or Trojan-horse attacks [AUCS96, FOSO96].

Privileged Users: Adversaries in this category have direct access to the host where the target program is running. In addition, they may have the following privileges.

- Access to private memory of other user or system processes
- Access to source code of the target program
- The ability to introduce and execute random software on the host

- The ability to manipulate and replace system software

Read access to the host memory implies that the adversary can obtain the binary image of a loaded executable program. That includes code as well as data associated with it. Write access gives the perpetrator the ability to modify main memory.

Access to source code of the target program suggests that software protection should not, and cannot hinge on the obscurity of the source program—determined adversaries will acquire copies of the original source program via out-of-bound means, and it is virtually impossible to guarantee such secrecy. However, a compromised source program does not necessarily imply direct compromise or immediate knowledge of the running executable program. That is, an executable program generated from a known source program, aside from being functionally equivalent, could contain extensive syntactic or semantics differences from the source program such that impersonation or intelligent tampering of the binary would still require analysis of the executable program. This premise is the cornerstone of much of this work, and we will elaborate in later sections why the premise stands and how it can be exploited as a basis to devise software protection mechanisms.

The ability to introduce and execute random software implies that the intruder may have access to specialized software analysis tools such as debuggers, decompilers and system diagnostic utilities. They can perform analysis on-line such as system diagnostics, or off-line such as black-box testing, execution emulation, and break-point-based debugging.

The ability to manipulate and replace system software suggests that the host security mechanisms such as those provided by the operating system can be compromised. Therefore any mechanism deployed to protect the software should not depend on the authenticity or security of the host OS. This assumption is, of course, the most troublesome—once you allow the host OS to be compromised, the perpetrator may have near complete control of the platform, and their action henceforth is only limited by available resources.

There is, however, one restriction on the intruder capabilities—he or she may not substitute or install hardware on the host system. Altering hardware configurations requires physical access to the host system. It is reasonable to assume that such access is discouraged, or to a large extent, difficult to obtain. This assumption eliminates the possibility of special hardware analysis tools such as bus analyzers or hardware monitors deployed directly on the target host. It should be noted that the use of external hardware for off-line analysis is still a possibility.

At this level of sophistication, the adversary has access to ample system resource and a great deal of knowledge on how the system works. Security attacks from these adversaries are the most powerful and also the most difficult to defeat. In fact, no security mechanism exists and none could be developed that will provide protection against such adversaries in the absolute sense—there may not be any workable solution against perpetrators with unbounded resources. What we aim to do in this work is to:

- Increase the technical difficulty to deter security attacks from malicious insiders and privileged users, and
- Understand and provide a theoretical basis to determine exactly what returns each protection

mechanism provides in order to make informed decisions.

3 EXPLORING THE SOLUTION SPACE

In this section, we explore the solution space for the problem described in the previous sections. A realistic solution for the software protection problem is to raise the technical bar for launching a feasible attack, i.e., the majority of the impersonation or tampering attacks must be either computationally or economically infeasible to accomplish. With this goal in mind, we explore a complexity-driven solution space.

The basis for judging technical difficulty is the complexity of computation involved in the operation. The notion of *computational complexity* here is used in a slightly unconventional sense. For example, computational complexity, in the traditional sense, is dominated by the order of growth of the algorithm—an operation that is of the order $O(n)$ is considered more efficient and less complex than one that is of $O(n^2)$. This is regardless of the lower order terms of the running-time formula and the constant coefficient of the leading term that determines the order of growth. In this work, it is not the order of growth alone that is of interest. Instead, what of interest is the *operational complexity*, affected by the input size, the constant coefficients and the order of growth. For instance, if the complexity of an attack algorithm can be expressed as

$$an^x + bn^{x-1} + \dots + c,$$

with n being the input size, a feasible defense mechanism might aim to raise the order x , the input size n , or the most significant coefficient a .

The discussion in section 2 states that intelligent tampering and impersonation attacks require certain information about the program. This information is the identification secret (or secrets) that constitutes the basis of the program's identity (without this secret, the trusted server cannot differentiate between a legitimate program and an imposter). The identification secret can appear in the form of protocols, cryptographic keys, or a particular set of behavior patterns that can be verified by the trusted server. For the purpose of this discussion, we assume that the target program carries such identification secrets, and that there exists a mechanism for a trusted server to authenticate the secrets and verify to whom it is communicating.

The presence of such an authentication mechanism implies that impersonation or tampering is impossible without performing program analysis to determine what the secret is, and how it will be verified. For example, consider the following code sequence:

```
int a = function1( );
int b = function2( );
Check_for_intrusion(&a, &b);
...
p      = &a;
...
Integrity_check(p);
```

If an adversary were to tamper with the `Check_for_intrusion()` function, they need to under-

stand whether and how the `Check_for_intrusion()` function changes the values of `a` and `b`, and whether the value of `a` and `b` are used later in the program. Without this knowledge, the adversary's action might be revealed when `Integrity_check(p)` is called.

In general, an adversary aiming to tamper with the program in an intelligent way must understand the effect of his actions, and this boils down to an understanding of the program semantics. One way this understanding can be acquired is through program analysis, and thus the operational complexity of intelligent attacks is determined principally by the complexity of program analysis. The solution framework used throughout this work incorporates various techniques to increase the complexity of program analysis, and thereby decrease the economic incentive of the attack.

Before discussing techniques to obstruct program analysis, we present a model of complexity for program analysis. The purpose of program analysis is to obtain information, and the complexity model is therefore established based on the information being analyzed. The model has three dimensions:

- **The amount of information:** Intuitively, program analysis is more complex the more information that must be analyzed. For example, attacking the network-wide survivability mechanism requires the compromise of a collection of monitor programs at different locations. If the monitors are diverse, independent analysis efforts will have to be expended to analyze each monitor. More effort (hence complexity) is required than would be required if all monitors were identical.
- **The computation of analyzing the information items:** There is a cost in complexity in analyzing each information item. This computational complexity is a significant factor in the overall difficulty of the program analysis.
- **The information lifetime:** Operational complexity, when rated against available resources, can be measured in terms of time. Increasing the amount of information and the complexity of analyzing the information can be viewed as efforts to increase the time required for an attack, while limiting the lifetime of the information serves as a complimentary tactic—it imposes a time bound within which the attack must be completed. The shorter the information lifetime, the more resources are required for the analysis, and hence the more difficult the attack.

These dimensions determine the complexity of program analysis. Using this model, techniques accentuating one or more of the dimensions present increased difficulties in program analysis. The solution framework used throughout this work is based on this complexity model, and is comprised of techniques that yield varying degrees of complexity along the different dimensions. The components of the solution framework include the following elements:

- **Diversity:** The notions of temporal, spatial, data, and design diversity are explored to introduce complexity and variations in the program. The diversity techniques present means to increase the analysis complexity as well as limit the time window of attacks.
- **One-way Translation:** One-way Translation is a compiling process in which source programs are translated into functionally equivalent but structurally varied binaries. The resulting binary programs incorporate properties that are difficult to analyze. When combined with temporal diversity, this mechanism provides a powerful way to obstruct program analysis. The

translation process is driven by a random number, and therefore is difficult to invert.

- **State Inflation:** This includes a set of mechanisms to obstruct dynamic program analysis techniques such as black-box testing by inflating the program state space.

3.1 Diversity

Diversity is an important engineering principle in building dependable systems. For example, in the design of an aircraft, *geographic* diversity is often used in the layout of hydraulic lines—each of the redundant lines feeding control surfaces pass through different parts of the fuselage and wings. This design helps to ensure dependable operation by tolerating certain perturbations in the environment including various forms of physical damage.

Incorporating diversity into the design of secure systems helps to reduce vulnerabilities that arise from uniform designs that are often the source of replicated flaws [FOSO96]. Four forms of diversity—*spatial*, *temporal*, *design*, and *data* diversity—are particularly useful in securing software execution. These forms of diversity have the following meanings in the context of this research:

a) Spatial Diversity: Spatial diversity refers to the placement of different instances of the same software at different locations. In a network, this placement refers to the use of different addresses (in any address space) on different network nodes.

The principles of spatial diversity can be used to thwart class attacks, a type of security attack based on exploitation of the same software or configuration flaws. For example, most script-driven attacks capitalize on a particular set of known flaws, and the same attack may be attempted on thousands of computer systems. If the different program instances are placed at different physical locations throughout the system, an intruder must invest more effort if the goal is to corrupt the network-wide survivability mechanism. Spatial diversity increases the amount of information an intruder must analyze to launch an attack, and is especially important considering that a large number of known security attacks are class attacks [AUCS96, FOSO96].

b) Temporal Diversity: Temporal diversity refers to a periodic variation in the software characteristics over time. Temporal diversity serves as a means to limit the lifetime of information, and hence the time window for a particular attack. If a successful attack takes longer than the lifetime of the information, then clearly it will not succeed.

As an example, suppose that after obtaining the binary image of an executable program P , an intruder attempts to perform a systematic state-space search to reverse engineer the binary program. If this effort succeeds after time period ΔT , the result might be a successful tampering or impersonation attack against P . However, if the properties of P change within ΔT ; i.e., if P is replaced with P' , the information obtained at the end of ΔT might prove to be ineffective if used against P' .

Temporal diversity implies dynamic changes—a property or a data element may only be valid or have security-related consequences for a limited time. In this work, temporal diversity is realized with periodic replacement and reorganization of the binary program and its properties.

c) Design Diversity: Design diversity is the use of different designs within several programs that

implement the same specification. It has been employed in various forms of fault-tolerant software including recovery blocks and N-version programming.

Design diversity holds promise in the security area addressed in this research because of the possibility of detecting tampering in a subset of the versions by observing differences in outputs. Multiple versions of the software to be protected would be written with each one prepared by an independent team. At execution time, all of the versions would be executed in such a way that their outputs could be compared with any deviation from the majority indicating a fault of some sort, possibly tampering.

d) Data Diversity: Data diversity employs multiple copies of the *same* program operating on different data. In a process called data re-expression, the data that the copies use is intentionally transformed in such a way that the output of the software is either identical or almost so for each version of the data.

Data diversity might be employed in protecting trusted sensor software by running multiple copies of the trusted software and checking that each yields the same (or properly related) data for the trusted server. In this way, any tampering would have to affect all versions essentially simultaneously in order to be effective.

3.2 One-way Translation

One of the principal program analysis techniques is static analysis—a technique to analyze program properties by examining the static image of the binary program. Static analysis can reveal program properties such as uses of variables, data locations, etc. This information can then be used in targeted tampering of the program.

A comprehensive static analysis on a program requires, as a minimum, the following information:

- Control-flow information
- Data-flow information

Control-flow information provides knowledge on the program flow control that constitutes the basis of further analyses.

Data-flow information provides knowledge about data quantities in a computer program such as the possible “modification, preservation and usage” of these quantities [HECH77]. Examples of data quantities include variables, instructions and memory locations.

The complexity of static analysis, therefore, depends on the complexity required to acquire the control flow and data flow information. The goal of One-way Translation is to incorporate techniques to obstruct control flow and data flow analysis. Some of the techniques discussed in this research are:

- **Masking control flow:** The control flow of the program can be masked by insertion and restructuring of control constructs. By adding non-functional code and breaking and reorganizing existing control constructs, the program control flow can become arbitrarily complex.

This is designed to obstruct control-flow analysis, a necessary step in decompiling or reverse engineering of programs.

- **Masking code or data content:** Data representations, as well as code constructs, can be restructured in such a way that it will be difficult to recover its original content or even its intent. For example, variables can be divided into subparts, and computations on the variable could be replaced with corresponding computations on the subparts and an operation to construct the correct result. Similar techniques can be applied to arrays, statements and subroutines.
- **Masking code and data location:** Some flow analysis techniques rely on code generation conventions such as the placement of local variables, etc. This type of analysis can be thwarted by the use of random code or data allocation algorithms. Furthermore, certain types of restructuring, when applied to both instructions and data, can also be used to obfuscate code and data locations. For example, a function can be in-lined at its call point or restructured to an arbitrarily different signature.
- **Masking data usage:** One of the primary functions of data flow analysis is to determine the usage of data – when and where they are used in the program. Data usage provides critical information for code optimization, and in this context, it can be used to facilitate intelligent tampering. Data aliases, for example, can complicate the analysis of data usage. Similarly, the use of indirect addressing and pointer manipulation can also be used to mask information on data usage.

These techniques aim to obscure information contained in the program. The premise is that by obfuscating, the resulting program will be more difficult to analyze, thus more difficult to decompile or reverse engineer.

It is important to raise the question of different objectives between conventional program analysis and what is intended in this work. The former has the goal of code improvement, thus a more aggressive, global analysis tactic is desirable. The latter, however, intends to gain specific knowledge to allow targeted program manipulation, and it therefore does not necessarily require as ambitious an analysis strategy. While that may prove to be true in some cases, the ultimate objective of this work is to make the task of program analysis as difficult as possible. In other words, the techniques employed here must include an effort to force the use of the most advanced analysis techniques possible. For example, disseminating critical information throughout the entire body of the program requires a global analysis to gather the necessary information. In so doing static analysis will be ineffective if not used in its most aggressive form, or not applied to the entirety of the program.

3.3 State Inflation—Increase the Complexity of Dynamic Analysis

Dynamic analysis such as black-box testing analyzes the program behavior without delving into the internal details of the program. The goal of this type of analysis is to gain insights into the semantics of the program behavior in order to emulate it. For example, testing the program with different input parameters and observing its output behavior may reveal information that can help determine the program state space.

Dynamic analysis can be conducted via off-line execution and testing. Alternatively, an intruder can attempt black-box analysis by observing the legitimate execution and using Markov analysis-like techniques to infer the relationships between program output and past events in the environment [INGE71].

If the state space of the program regarding input and output is simple, with relatively low effort the intruder can deduce the state space and emulate it to impersonate the legitimate program. For example, consider a monitor program for which there are three basic input states: UP, DOWN, and DEGRADED, and the program outputs an integer 0, 1, and 2, respectively, for each of the input states. In this case, a simple black-box testing would suffice in revealing the entire state space.

To protect against dynamic analysis, we propose the technique of *state inflation*. The purpose of state inflation is to increase the complexity in the state space of the program such that dynamic analysis would provide poor returns on the investment of time and effort. Again, the effectiveness of the scheme is measured in terms of the amount of information an intruder might be able to gather within a prescribed time frame.

The benefit of state inflation is perhaps best illustrated with an example. Consider again the example of a monitor program which operates on three basic input states: UP, DOWN, and DEGRADED. The program generates output integers 0, 1, and 2, respectively, based on the input state. Now consider instead of generating one of the three integers, the monitor applies the following algorithm to generate series of number x 's such that:

$$E(k, x1) \bmod 3 = 0$$

$$E(k, x2) \bmod 3 = 1$$

$$E(k, x3) \bmod 3 = 2$$

$E(k, x)$ is a one-way function, and k represents a key that the monitor shares with the trusted server. Instead of transmitting 0, 1, or 2 across the network, the monitor transmits a randomly chosen x 's in the appropriate series of numbers (e.g. $x1$ for UP, $x2$ for DOWN, and $x3$ for DEGRADED). The receiver of the integer can then easily compute:

$$E(k, x) \bmod 3$$

to obtain the state information, while an observer, not knowing k , will not be able to determine which x 's correspond to which state.

While this example is reminiscent of encryption, what is important here is that the one-to-one mapping between the input states and the output integers are replaced with a one-to-many mapping. Note that there is an arbitrarily large number of output values for each input state, which will appear essentially random to an outside observer.

Dynamic analysis is based on information obtained by observing program execution; that is, each state transition during the program execution disseminates certain amount of information into its

environment, and over time the aggregate of this information may be sufficient for an observer to determine the entire state space of the program. What state inflation attempts to do is to expand the program state space and the number of possible state transition for the same operation. Consequently, the average amount of information (i.e. the entropy of information) provided by each state transition will decrease. More effort thus must be expended to gather an equivalent amount of information.

4 ONE-WAY TRANSLATION

In this section we describe a design for a compiler-based mechanism to achieve program diversity and complexity, and in turn to obstruct static analysis of programs. This mechanism, by the name *One-way Translation*, uses compilation techniques to generate binary programs that are resistant to static analyses.

One-way translation transforms a source program into a binary program in such a way that the reverse transformation cannot be determined without the expenditure of tremendous resources. Formally, One-way Translation can be described as follows:

Let TR be the translation process, such that $P \xrightarrow{TR} B$ translates a source program P into a binary program B . TR is a one-way process if the time taken to reconstruct P from B is greater than a specific constant T .

The core of One-way Translation is the semantically-equivalent transformation of programs to incorporate design diversity and code complexity. When combined with temporal diversity techniques (e.g., a periodic replacement of the program), it provides the necessary elements to deter program analysis, and ultimately defend against intelligent tampering and impersonation attacks.

4.1 A Model of Semantics-Preserving Transformation

We first present a model of semantics-preserving transformation since it departs from the traditional meaning of functional equivalence.

In traditional compiler parlance, *semantics-preserving* transformations preserve the input-output behavior of the program. In other words, the program, before and after the transformation, must produce the exact same results if given the same input [MUCH97].

It should be noted that this traditional definition of functional equivalence is often violated in the actual practice of compilation. For example, the result of commutative operations such as addition, in theory, does not depend on the order of the operation being performed. However, reversing the order of a commutative operation can lead to different results due to possible rounding errors [ref?].

In this work, we employ a relaxed notion of functional equivalence. Function is not defined in terms of input-output relations. Instead, it is defined as a set of high-level specification of tasks. Different implementations, if they fulfill the tasks specified, are considered semantically-equivalent irrespective of their input-output behavior. For example, if a program's function is to report the temperature of the day, two different algorithms—one reporting the temperature in Fahrenheit and the other reporting in Celsius—both accomplish the specified task, and are therefore consid-

ered functionally equivalent despite the fact that their input-output behavior is quite different.

Under this definition of functional equivalence, code transformations may affect the internal structure as well as its external observable behavior of a program. Transforms affecting the external behavior of the program may very well alter the program signature. Also note that the traditional definition of functional equivalence is subsumed by the new definition; that is, program transforms that are considered semantics-preserving in the traditional sense are clearly semantics-preserving under the new definition.

The new notion of functional equivalence is application specific—the set of semantics-preserving transformations for one application may or may not preserve functional for others. For example, replacing a DES encryption algorithm with an implementation of RC4 is a functionally equivalent transformation for the purpose of encryption, but would be meaningless where encryption is not concerned. The specification of tasks will have to be derived from the domain knowledge of the application.

The notion that there exists an equivalence class of programs that differ not only in terms of internal representation but also in external behavior is fundamental to the One-way Translation idea. This equivalence class embodies the idea of software design diversity. Recall the discussion of temporal and spatial diversity in the previous sections, and observe that spatial diversity can be achieved by deploying different programs within the same equivalence class at different physical locations, and temporal diversity can be realized by updating the program periodically with copies in the same equivalence class.

4.2 The One-way Translation Compiler

The One-way translation process is based on the idea that the representation of a program, both high-level and intermediate, can be modified to incorporate semantics-preserving transformations. The program modifications are embedded in the compilation process. They constitute the mechanism via which design-diverse programs within the same equivalent class are generated.

The code transformations can be introduced in various stages of the compilation process. However, since one of the goals of transformation is to generate different executable programs, it is advantageous to perform the modification when the memory layout and the instruction stream of the program have not yet been determined. In this work, we choose to perform code transformations at the source and the intermediate representation levels.

It should be noted that functionally-equivalent transformation can also be applied to binary programs directly [CER99, KEHO97]. However, only limited forms of transformation are applicable to binary programs, and the issue of platform-dependence could hinder the wide adoption of the scheme. For these reasons, we choose not to implement code transformation at the binary program level.

The specific transformations applied in each compilation are determined by a random number. Consequently, the One-way Translation process is capable of generating a suite of different programs based on the same original source program. Recovering the original source program from a binary version is “difficult” since it requires knowledge of the randomness in the translation process.

In order to provide the code variation and complexity, two types of modification can be applied to a target program: internal transformations, and behavioral transformations.

4.3 Internal Transformations

This type of transformation affects the internal representation of a program. Such transformations may or may not influence the program's external behavior directly. For example, changing the order of non-interfering instructions alters a binary program's internal structure, but does not affect the result of execution.

Internal transformations can be divided into the following two categories:

- **Control-flow Transformation:** Control-flow transformation modifies an existing program control-flow to that of a more complex form in order to deter static analysis—most static analyses rely on information about the program control-flow to obtain analysis precision.
- **Data-flow Transformation:** The problem of extracting a secret (or secrets) from a binary program can be reduced to the equivalent problem of conducting a data-flow analysis. Data-flow transformations can be applied to complicate the process of analyzing the modification, preservation and usage of data quantities, the basis of most data-flow problems.

4.4 Behavioral Transformations

This type of transformation alters the observable behavior of a program. The following categories of transformation can be applied to the program in order to provide variability in its behavior pattern:

- **Interfacing Protocol Change:** The program interfaces with the trusted servers via a predetermined protocol. This protocol is initialized at each installation to a unique instance. This of course implies changes in both communicating parties—the interfacing program in the trusted server must be changed accordingly.
- **Change of identification secrets:** The program's identification secret such as a cryptographic key can be made installation unique; that is, the one-way translation process generates a unique key for each binary program. At each installation, the program is updated with a functionally-equivalent version with a new key.
- **Change of input-output behavior:** The example in section 3.3 illustrates one scenario in which the input-to-output relation of the program is changed from a one-on-one mapping to a one-to-many mapping. This modification changes the input-output behavior from the outset, but still preserves the meaning of the program.
- **Alternative implementation of function:** This is very much a software component plug-n-play idea. This work relies on user specification as well as alternative implementations supplied by the user to accomplish the plug-n-play model.

In the following sections, we detail the code transformations in the above categories and present a number of example techniques.

5 INTRA-PROCEDURAL CODE TRANSFORMATIONS

Static analysis at the intra-procedural level analyzes information within a particular function. This type of analysis can be classified into two categories: *flow-sensitive* and *flow-insensitive* [BANN79, MARL93]. Flow-sensitive algorithms consider program control-flow information and, in general, yield more precise results than flow-insensitive algorithms. Flow-insensitive algorithms do not make use of control-flow information during the analysis, and therefore must settle with a solution that summarizes over all possible control flow paths. For this reason, flow-insensitive analysis is generally more efficient with the price of being less precise.

It is important to note that control-flow analysis is the first analysis stage—it provides information about the program control transfer that is essential for subsequent data-flow analyses. Without this information, any data-flow analysis is restricted to the basic-block level only and will be fundamentally ineffective for programs where data usage is dependent on program control-flow.

The primary purpose of the intra-procedural code transformations is to conceal the program control-flow and thereby hinder both control-flow and data-flow analysis. The end result of these transformations is as follows:

- Flow-sensitive analysis cannot be more precise than flow-insensitive analysis.
- Flow-insensitive analysis is made ineffective by incorporating data whose usage is control-flow dependent.

In this section, we first present the fundamentals of control-flow analysis and how they are used in this work to conceive anti-static-analysis techniques. We then describe a list of code transformations that are applied in this work.

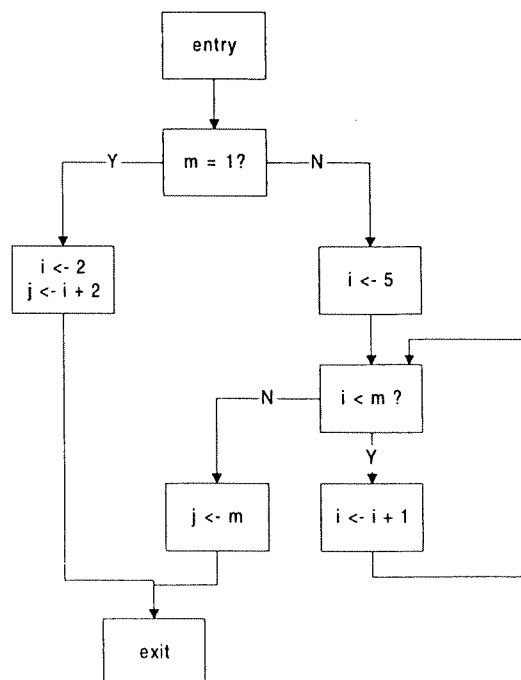


Figure 2. An Example Flowgraph

5.1 The Fundamentals of Intra-procedural Control-flow Analysis

Control-flow analysis encodes and makes explicit the flow of control in a program. Intra-procedural control-flow analysis constructs the flow graph for each procedure as follows—consecutive statements within the procedure are partitioned into *basic blocks* such that once the first statement of the block is executed, all statements in the block are executed sequentially. Program control is transferred to another block once every statement in the current block has been executed.

A flow graph represents the flow of control among basic blocks. Formally speaking, a flow graph is a triple $G = (N, A, s)$, such that N is a set of vertices representing basic blocks, A is a set of arcs between blocks, and s is the starting vertex in the graph. An arc(x, y) from node x to node y indicates that program control can transfer potentially from block x to block y . There exists at least one path from the starting node to every other node in the graph. Figure 2 shows an example of a flow graph and the corresponding code segment.

Real-world programs tend to have control-flow that can be easily discerned since this is encouraged for program clarity and enforced by high-level language constructs. In such a program, branch instructions and targets are easily identifiable. Thus determining the program flow graph is a straightforward operation of complexity $O(N)$, where N is the number of basic blocks in the program.

Now consider the case where branch instructions are indirect jumps whose target addresses are not known at compile time. Figure 3 shows such a code segment in pseudo assembly code. In this example, the instruction at `s12` is an indirect branch instruction whose target is defined in register 1. In order to determine to which location this instruction will branch, a static analyzer will have to examine the code to reveal that the content of register 1 is defined at `s1` (the definition at `s1` overwrites the one at `s0`). What just happened here is a *use-and-def* analysis in which a *use* of a variable (whose content held in register 1) is identified and its latest *definition* (at `s1`) is found [MUCH97]. The dashed line in Figure 3 illustrates the use-and-def chain information.

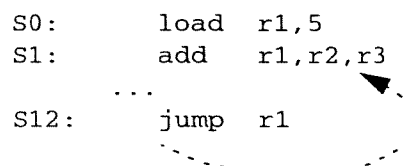


Figure 3. Example Code Segment for Indirect Jumps

Such a use-and-def data analysis is necessary in determining the precise program flow when branch targets are data dependent. It is then clear that, in this case, building the program flow graph is at least as complex as performing the necessary data-flow analysis to resolve indirect branch targets.

It is widely known that many data-flow problems do not have efficient solutions for programs with certain characteristics such as general aliasing [MUCH97]. Some problems have proved to

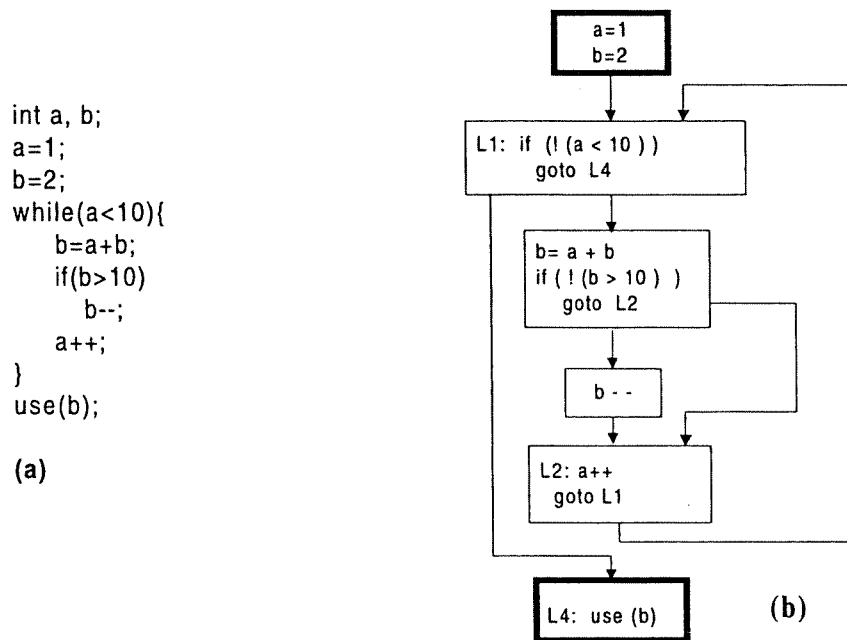


Figure 4. Dismantling of High-level Control Constructs

be NP-complete [LAN921, LAN922, MYER81]. This difficulty in data-flow analysis constitutes the basis of the approach to obfuscation outlined in this section. More precisely, the strategies employed in this work to defeat static analysis are as follows:

- Transform the original program control-flow to that of a data-dependent nature. In other words, the control-flow analysis is transformed into a data-flow problem.
- Increase the complexity of the data-flow analysis to determine the branch targets by incorporating certain program characteristics such as non-trivial aliasing.

The rest of this section elaborates on the code transformations that implement these strategies.

5.2 Control-Flow Flattening

To make the program control-flow data dependent, we employ a set of code transformations called “control-flow flattening”. These transformations are performed in two steps. In the first step, high-level control structures are decomposed into equivalent *if-then-goto* constructs. Figure 4 illustrates such a transform.

In the second step the *goto* statements are modified such that the target addresses of the *goto*’s are determined dynamically. At the source-program level, this transform is modeled by replacing each *goto* statement with an entry to a *switch* statement. The switch control variable is assigned dynamically in each code block to determine which block is to be executed next. An example of such a transform is illustrated in Figure 5.

With these transformations, direct branches are replaced with data-dependent indirect jumps. As a

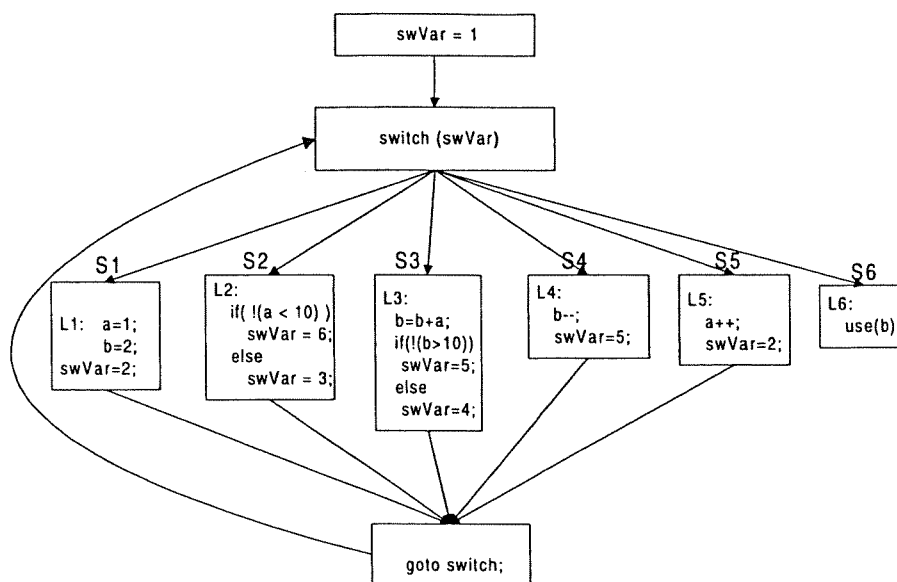


Figure 5. Transform to Indirect Branches

result, the flowgraph that can be obtained from static branch targets degenerates to a form shown in Figure 6. We will refer to such a degenerate flowgraph informally as *flattened*.

5.3 Introduction of Aliases

After flattening of the program control flow, building the program flowgraph hinges on the complexity of determining branch targets, which is in essence a *use-def* data-flow problem.

In the example shown in Figure 5, the values of the switch control variable `swVar` are assigned dynamically with the constant assignment statements. A constant propagation analysis [MUCH97] combined with use-and-def data flow on the value of `swVar` will quickly reveal, for each block, what the branch target is for the next block, and consequently reveal the entire con-

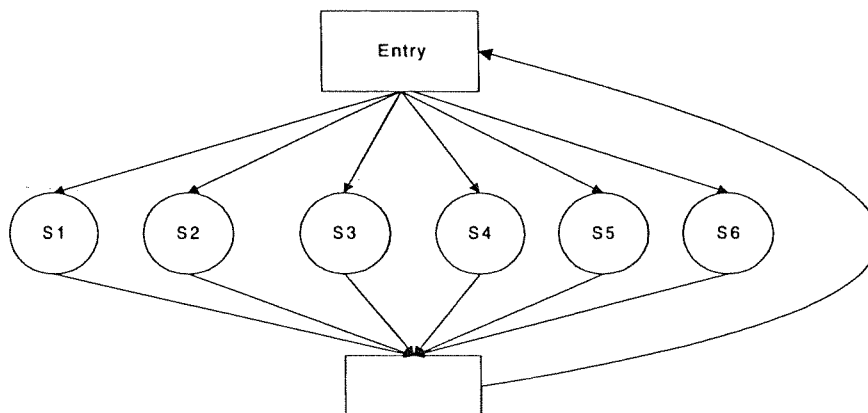


Figure 6. A Flattened Flowgraph

control-flow graph. To further complicate the static determination of the branch targets, two additional modifications to the program are made: index computation, and introduction of aliases.

Index Computation: Consider the code segment in Figure 7(a). A use-def analysis on the value of `swVar` (contains branch target information) is straightforward (the dashed line indicates the use-def information chain). Now consider the code segment in Figure 7(b) in which a global array `g[]` is introduced and the value of `swVar` is computed through the elements of the array. Replacing the constant assignments in Figure 7(a) with complex expressions involving array elements implies that the static analyzer must first deduce the array values before the value of `swVar` can be determined.

Some of the array elements contain constant values that are used in the computation of switch variables, while others simply contain arbitrary values. The array elements themselves do not have to remain constant—assignments to the array can be made throughout the program execution as long as it does not interfere with the branch target computation.

For example, consider the scheme in which every n th element in the array contains a data value x such that:

$$x \equiv 1 \bmod j$$

where j is a value contained in a specific location in the array (this location can change with every compilation). It is easily seen that any numeric value (such as array subscripts) can be computed using these data values that are n elements apart. Throughout the course of execution, the values of the array can be overwritten and j can be updated accordingly so that branch target computation can continue undisturbed.

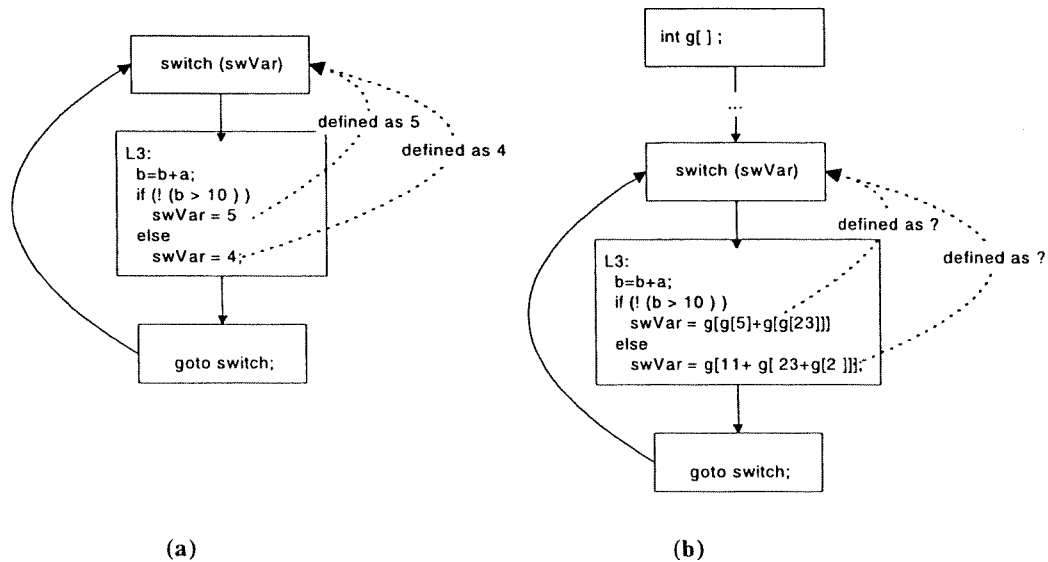


Figure 7. Example Illustrating Dynamic Computation of Branch Targets

Aggregates, such as arrays or complex data structures, are difficult for static analyzers to process [MUCH97]. Statically analyzing data quantities that have aggregate elements as their values is particularly difficult when elements of the aggregates do not remain static throughout execution. Most static analyzers simply assume that a reference to any element of the aggregate is a reference to the entire structure, and that, potentially, all elements can be changed if an assignment is made to any element. This is the safest and most conservative position.

An algorithm whose purpose is software tampering rather than code improvement would most likely not take this conservative stance. An aggressive data-flow algorithm that is specially tuned to analyze arrays can provide more precise information about arrays and their operations. To deal with the more sophisticated analysis algorithms, the next set of code transformation makes use of a program property that is a primary contributory factor in the complexity of data-flow analysis. This property is aliasing.

Many classical data-flow problems have been proven to be NP complete [LAN922, MYER81]. A fundamental difficulty that data-flow analysis must deal with is the existence of aliases in the program. Precise alias detection in the presence of general pointers and recursive data structures is known to be undecidable [LAN921], and that is the key reason why any data-flow problem influenced by aliasing is fundamentally difficult.

Our second set of transformations focuses on the introduction of non-trivial aliases into the program to influence the computation, and hence the analysis, of the branch targets. These transformations are as follows:

- We first introduce an arbitrary number of pointer variables in each function (this can be parameterized).
- We then insert artificial basic blocks, or code in existing blocks, that assign the pointers to data variables including elements of the global array whose values are used in computing the branch targets.
- Now we can replace references to variables and array elements with indirection through these pointers. Previously meaningful computations on data quantities can be replaced with semantically equivalent computation through the pointers.
- As much as possible, uses of the pointers and their definitions are placed in different blocks. More importantly, assignments to array elements will appear as assignments to the pointer variables which are aliased to array elements.

Some of the basic blocks will execute in all traces of the program, and others are simply dead code. Since the static analyzer does not know which blocks actually execute, and since the definitions of the pointers and their uses are placed in different code blocks, the analyzer will not be able to deduce which definition is in use at each use of the pointer—all pointer assignments will appear live.

Recall the notions of flow-sensitive and flow-insensitive analysis. A flow-insensitive analysis does not depend on the program control-flow information; that is, the analysis perceives the program as a collection of basic blocks, the inter-relations among which are ignored.

It can be shown that the flattened flowgraph in Figure 6 is equivalent to the control-flow perceived by a flow-insensitive analysis [HIND99]. Without knowledge of the branch targets and the execution order of the code blocks, a flow-sensitive analysis cannot provide better precision than a flow-insensitive one.

6 INTER-PROCEDURAL CODE TRANSFORMATION

6.1 Program Call Graph and Inter-procedural Alias Analysis

To compute the effect of aliasing precisely, analysis must be performed at the inter-procedural as well as the intra-procedural level. This is because function invocations can affect aliases in both the calling and the called function. For example, if a global variable is passed as a parameter in a function call, the global variable and the corresponding formal parameter would become aliases inside the called function. Similarly, if an assignment statement inside the called function assigns the address of a global variable to a pass-by-reference parameter, the actual parameter and the global variable will become aliases in the calling function, immediately after the return from the called function.

An inter-procedural data flow analysis relies on the function invocation relationship to determine, among other things, the static information propagation paths in the program, which can then be used to reason about alias relations resulted from function invocations.

The function invocation relations can be encoded in a graph called Program Call Graph (PCG). Formally, a PCG is a triple (N, e, p) where; N is the set of functions in the program such that $N = \{p_1, p_2, \dots, p_n\}$, p is the function that contains the program entry point; and e is a set of directed edges such that if (p_i, p_j) is an element of e , there exists at least one call from function p_i to p_j .

Figure 8(a) shows an example of a program skeleton in which function f calls g , g calls h and i , and i calls j and g . The PCG for this program is shown in Figure 8(b). The entry function f is marked by the double circle.

Construction of the PCG is a straightforward process when all function calls are explicit—a simple textual pass over the program will suffice. In the presence of function pointers (or function parameters and function variables), however, a call-site may not be bound to a unique function statically, and therefore the process of constructing the PCG can be more complex.

Several approaches to build the PCG in the presence of function pointers exist, each with varying degrees of complexity and precision [EMAM94]. An approach that simply assumes that an invocation through a function pointer may invoke any function in the program requires a single pass over the program, and thus is the least costly with the least precision. An approach that takes into account only the functions that have been instantiated requires a flow-insensitive traversal of the program, and it generally provides better precision.

A more precise PCG can be obtained by conducting pointer alias analysis prior to building the

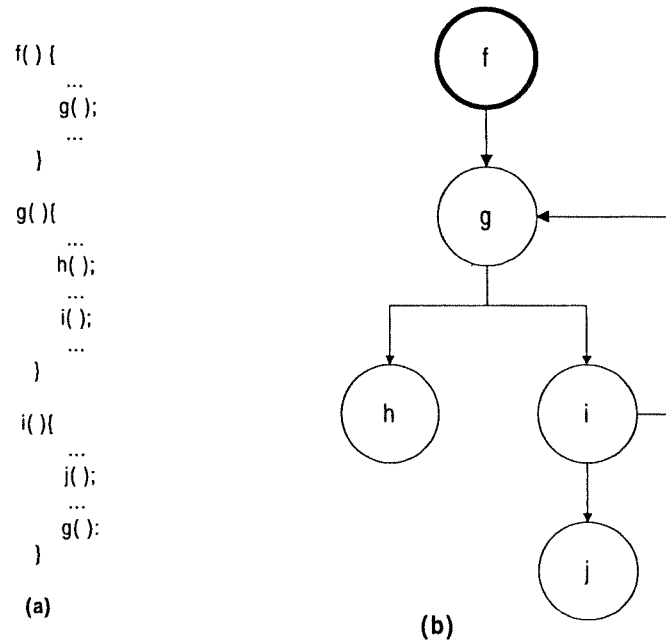


Figure 8. Example illustrating Program Call Graph

final PCG. The pointer analysis can help to restrict the number of functions invocable from a call site to the set of functions that are deemed to aliased to the function pointer at that particular point in the program. In other words, construction of a precise PCG hinges on none other than alias analysis of the function pointers.

It should be clear by now that if function pointers are treated in the same manner as other data pointers, the techniques described in section 5.2 can be applied to introduce aliases for function pointers. In so doing we reduce the precision of the PCG representation, and ultimately impede the process of inter-procedural data analysis.

This section describes a set of code transformations based on the above observation. The basic strategy behind these transformations is to modify function calls to include invocations through function pointers. In the mean time, alias-inducing techniques are applied to generate inter-procedural aliases as well as aliases to function pointers. The end result is increased complexity and reduced precision for inter-procedural control flow and subsequent data-flow analyses.

6.2 Function Call Transformations

The presence of function pointer makes the task of constructing precise PCG more difficult. Consider the code segment in Figure 9 in which `func1` calls `func2` and `func3`.

Now consider the code segment in Figure 7, which implements the equivalent functionality.

In Figure 10, `ptr`, `fptr1`, and `fptr2` are function pointers. To discern the target of the indirect call on line S6 in Figure 9, the analysis needs to determine the set of functions to which `fptr1` and `fptr2` are aliased. This requires knowledge of the alias relations that hold on entry to `func1` and possible changes to these relations up to line S6. Without this information, the function pointer `ptr`

```

S1:  func1( ) {
S2:      if ( x > 4 )
S3:          func2( );
S4:      else
S5:          func3( );
S6:  }

```

Figure 9. Example Illustrating Function Calls

on line S6 cannot be bound to any particular function at analysis time.

In this section, we present a set of code transformations designed to modify the program call structure to include invocations via function pointers. More specifically, the transformations include the following steps:

- Unify function signatures to a uniform signature (or a small set of signatures).
- Create function pointers and assign these pointers to the modified functions.
- Create function pointer aliases much in the same way described in the same manner described in section 5.2.
- Modify function calls to invocations through function pointers.

The following discussions consider these steps in turn.

6.2.1 Unify Function Signatures

An important part of the transformations is to create indirect invocations via function pointers. However, if each function in the program has a distinct signature, each invocation must be performed via a pointer of a distinct type¹. This in itself is not sufficient to fool a static analyzer of any intelligence, for it is a trivial task to match up invocations with different pointer variables. For this reason, prior to creating indirect calls, function signatures must be modified such that all functions in the program conform to only a small number of distinct signatures.

```

S1:  func1(fptr1, fptr2) {
S2:      if (x > 4)
S3:          ptr = fptr1;
S4:      else
S5:          ptr = fptr2;
S6:      ptr;
S7:  }

```

Figure 10. Example Illustrating Function Pointers

1. It is possible to use void pointers, but the invocation statement is still distinct, for each function might have different number and types of parameters.

Consider the two function invocations in *func1* in Figure 11. Both function *P* and *Q* return an integer. However, *P* takes one integer parameter while *Q* takes two parameters, one integer and one float. One viable method to unify those two function signatures is to modify *P*'s signature to add a float parameter as depicted in Figure 9. The bold letters indicate the artificial variable and parameter:

<pre> Func1() { int x, y, z; float f; y = P (x); z = Q (x, f); } </pre>	<pre> Func1() { int x, y, z; float f1, f2; y = P (x, f1); z = Q (x, f2); } </pre>
<pre> int P (int para1) { } </pre>	<pre> int P (int para1, float para2) { } </pre>
<pre> int Q (int para1, float para2) { } </pre>	<pre> int Q (int para1, float para2) { } </pre>
(a)	(b)

Figure 11. Function Signatures Before and After Modification

It should be noted that the number of distinct function signatures in a program is an application specific choice. As an extreme, all functions will have a single uniform signature in which case only one type of function pointer is required. Alternatively, functions can be grouped into a small number of groups, each with a distinct signature. The advantage of the latter scheme is efficiency; In general, a large portion of the functions in a program will contain a short list of parameters with common types (e.g. integers or floats). It is then more economical to group these functions together to form a common signature since the new signature may not be far from any of the original ones, and thus will have less an impact on the run-time space and time complexity.

A number of issues need to be addressed when modifying function signatures and the corresponding invocations. Some of those are discussed below:

- **Return type:** In order to unify functions with different return types, a *void* type is used in the transformed function signatures. An explicit cast back to the original type at the function invocation completes the transform.
- **Complex parameters:** Complex parameters such as structures, arrays or functions are replaced with void pointers. Consequently, both function invocations and references to the original parameters inside the function are modified as follows:
 - A direct reference to the original parameter inside the called function is replaced with an indirect reference via the void pointer parameter, following an explicit cast to allow the void pointer to point to the original parameter type.
 - At the function invocation, the original actual parameters are replaced with void pointers

which hold the addresses of the original parameters.

- Parameter list explosion: When unifying functions whose parameter lists do not intersect, it could lead to parameter list explosion. For example, unifying one function with all integer parameters and another one that has all floats will result in twice as many parameters in the unified signature. To avoid parameter list explosion, we impose an upper limit on the number of parameters a function can have. We also employ a naive algorithm to identify functions whose parameter lists heavily intersect with each other—these functions are good candidates for signature unification.
- These transformations result in a large number of functions with an identical signature. The extent to which existing function signatures are transformed is governed by parameters of the transforms (e.g. the number of distinct signatures). Observe that with these transforms, the same function pointer can refer to a large number of otherwise distinct functions.

6.2.2 Create Aliases with Function Pointers and Modify Function Invocations

When function invocations are made through function pointers, the number of potential functions to which a particular call site can be bound statically is equivalent to the set of functions the function pointer is aliased to at the call site.

Using the pointer manipulation techniques described in section 5.2, false alias relations can be introduced to the function pointers. False edges to the PCG, as perceived by a static analyzer, are

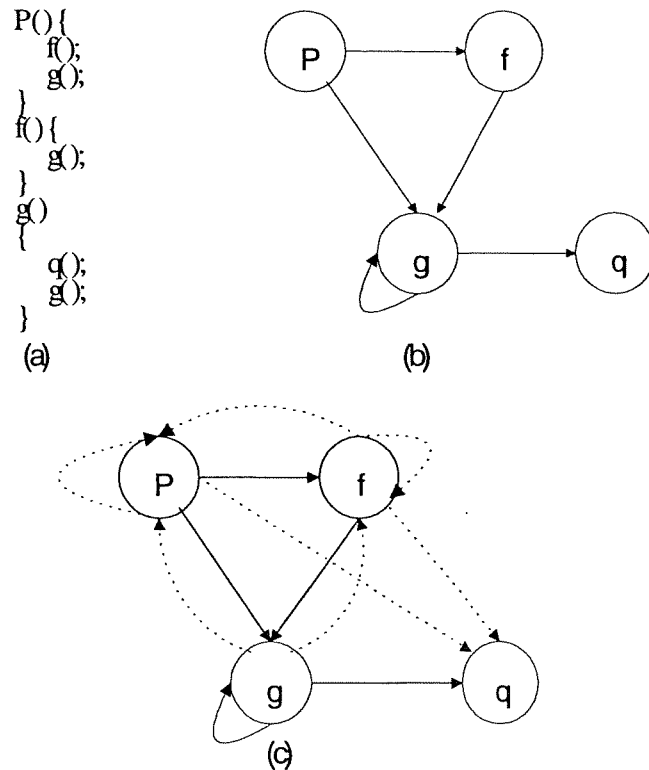


Figure 12. PCG with False Edges

therefore created. In the extreme, all functions can be altered to a single uniform signature. In such a case, the PCG will have a complete degenerate form; that is, any function that contains a function call will have an edge to every other node (function) in the PCG. Figure 9 illustrates such an example. The original call structure is in Figure 9(a) while its corresponding PCG is depicted in Figure 9(b). The post-transform PCG is illustrated in Figure 9(c) where the dashed lines are artificial edges added due to function pointer aliasing.

When a function pointer appears to be aliased to a large number of functions, statically it is impossible to tell exactly which function will be invoked and what data variables are being passed to the called function. This greatly reduces the precision of inter-procedural analysis whose purpose is to discern the information propagation paths among functions.

6.3 Inter-procedural Aliases

Under the conjecture that the PCG of the program is degenerate, and precise information propagation paths amongst functions are not retrievable statically. We describe, in this section, a set of techniques to further thwart static analysis by introducing inter-procedural aliases—aliases whose meaningful resolution requires none other than inter-procedural data-flow analysis.

Inter-procedural aliases are generally created by parameter passing and the accessibility of non-local stack locations. More precisely, the ways in which we introduce inter-procedural aliases are as follows:

6.3.1 Aliases in the called function resulted from the invocation

- Global and local reference aliases: When the address of a global variable or that of its alias is passed to a function, inside the function the global and the corresponding formal parameter become aliases.
- Parameter aliases: Aliases between parameters are introduced in several ways. For example, two pointer parameters become aliases of each other if the same address is bound to both of them, or one is bound to the address of a variable while the other is bound to the address of its alias.

6.3.2 Aliases in the calling function resulted from the invocation

- Alias through return values: If the called function returns the address of a variable visible in the calling function, and the return value is assigned to a pointer variable, the two variables become aliases upon returning from the invoked function.
- Alias through side effects: If the address of a pointer variable is bound to a parameter of a function call, and the parameter is subsequently assigned the address of another pointer variable, also visible in the calling function, the two pointer variables become aliases upon returning from the called function.

Function invocation can result in aliases in both the called and the calling function. This is because inter-procedural information propagation—there is a *forward* parameter binding process in which information propagating from the calling function to the invoked function results in aliasing in the latter. Similarly, there is a *backward* binding process in which information propagating from the called function back to its caller prompt new alias relations in the calling function.

In order to handle the information propagation, both forward and backward, it is crucial that the function invocation information is available in order to bind arguments and parameters properly.

7 THEORETICAL EVALUATION

In this section, we present a set of analytic results to show that the transformations presented in section 4 indeed produce the claimed complexity for program analysis.

7.1 An NP-Complete Argument

We have thus far conjectured that the difficulty of discerning indirect branch target addresses is influenced by aliases in the program. In this section, this claim is supported by a proof which shows that statically determining precise indirect branch addresses is an NP-complete problem in the presence of general pointers.

Theorem 1: In the presence of general pointers, the problem of determining precise indirect branch target addresses is NP-complete.

Proof: The proof consists of a polynomial time reduction from the NP-hard 3-SAT problem to that of determining precise indirect branch targets. This is a variation of the proof originally proposed by Myers in which he proved that various data-flow problems are NP-complete in the presence of aliases [MYER81]. Landi later proposed a similar proof to prove that alias detection is NP-complete in the presence of general pointers [LAN922].

Consider the 3-SAT problem such that

$$\bigwedge_{i=1}^n (V_{i1} \vee V_{i2} \vee V_{i3}), \quad (1)$$

where $V_{ij} \in \{v_1, \dots, v_m\}$, and $v_1, \dots, v_m$ are propositional variables whose values can be either *true* or *false*. The 3-SAT problem states that it is NP-hard to discern, in a general case, that whether the equation above is satisfiable.

The reduction is shown in the code below. The branch target address is located in the array element $A[*true]$. The *if* conditionals are not specified – the assumption is that all paths are potentially executable.

```

L1:  int *true, *false, **v1, **v2, ..., **v_m, *A[];
L2:  A[*true] = &f1();
L3:  if (-) v1 = &true;  $\overline{v_1} = \&false$ 

      else v1 = &true;  $\overline{v_1} = \&false$ 

      if (-) v2 = &true;  $\overline{v_2} = \&false$ 

```

```

else  $v_2 = \&true; \overline{v_2} = \&false$ 

if (-)  $v_n = \&true; \overline{v_n} = \&false$ 

else  $v_n = \&true; \overline{v_n} = \&false$ 

L4:  if (-)  $A[**\overline{v_{11}}] = \&f2()$ 

      else if (-)  $A[**\overline{v_{12}}] = \&f2()$ 

      else  $A[**\overline{v_{13}}] = \&f2()$ 

      if (-)  $A[**\overline{v_{21}}] = \&f2()$ 

      else if (-)  $A[**\overline{v_{22}}] = \&f2()$ 

      else  $A[**\overline{v_{23}}] = \&f2()$ 

      . . .

      if (-)  $A[**\overline{v_{n1}}] = \&f2()$ 

      else if (-)  $A[**\overline{v_{n2}}] = \&f2()$ 

      else  $A[**\overline{v_{n3}}] = \&f2()$ 

```

L5:

The code segment L1 declares the variables and an array $A[]$. $v_1, v_2, \dots, v_m$ are doubly dereferenced pointer variables. L2 assigns $A[*true]$ to the address of $f1$.

A path from L3 to L4 represents a truth assignment to the propositional variables for the 3-SAT problem. In this code, the assignment to *true* is represented as an alias relationship $\langle *v_i, false \rangle$, and the alias $\langle *v_i, false \rangle$ represents assigning *false* to variable v_i .

If the truth assignment for the particular path from L3 to L4 satisfies the 3-SAT formula, then every clause contains at least one literal that is true. This means that there exists at least one path between L4 and L5 on which the value of $A[*true]$ is never reassigned. Consider choosing the path that goes through the true literal in every clause, and in every clause it assigns $A[*false]$ to $f2()$ since every variable $*v_{ij}$ on that path is aliased to *false*.

If the truth assignment renders the formula not satisfiable, then there exists at least one clause, $(V_{i1} \vee V_{i2} \vee V_{i3})$, for which every literal is *false* (i.e., all the literals in the clause are aliased to *false*). This implies that $\overline{*v_{ij}}$ is aliased to *true* for this clause. Because every path from L3 to L4 must go through the following statement:

```

If (-)  $A[**\overline{v_{i1}}] = \&f2()$ 
    else if (-)  $A[**\overline{v_{i2}}] = \&f2()$ 
    else  $A[**\overline{v_{i3}}] = \&f2()$ 

```

Therefore, at program point L5, $A[*true]$ must point to the address of $f2$.

The above code segment shows that 3-SAT is satisfiable if and only if the branch target address contained in $A[*true]$ is the address of $f1$. This proves that 3-SAT is polynomial reducible to the problem of finding precise indirect branch target addresses.

7.2 The Parameters that Affect Alias Analysis

The NP-completeness proof presents a complexity measure for analysis of general case post-transform programs. However, such a proof does not guarantee the average case complexity—for any given program, the complexity of analysis could be well within the grasp of a static analyzer (because of its size, characteristics, etc.)

In this section, we examine the complexity measures for practical alias analysis. In practice, alias analysis is conducted in an approximation manner—the results reported by the alias analyzer may not be precisely the same as the actual alias relations, but usually contain a conservative estimate of the reality (a super set of the actual alias relations). How far the reported results are from the actual alias set is called the *precision* of alias analysis.

The discussion here is concerned with the efficiency of the alias analysis as well as its precision. An analysis algorithm may be able to reach a conclusion quickly if precision is not of major concern. A clear example is to trivially report that every variable is aliased to every other variable—an analysis would take very little time but would report with the least precision.

The complexity of alias analysis has been examined at a great length in various studies [HIND99, LAN922, LARY92]. In the discussion here, we distill from the previous research the essential parameters that affect alias analysis and explore the effect of the code transformations with respect to these parameters.

Before delving into the details of analysis, we introduce a number of conventions and terminology that are used in the subsequent discussions.

“Points-to” representation: The discussion hereafter is based on the *points-to* abstraction of alias information [EMAM94]. Using this representation, x points-to y if x contains the address of y . The “Points-to” representation is commonly regarded as more compact and efficient than the explicit alias pair representation [HIND99]. For example, the alias relations shown in Figure 10 can be represented with two points-to tuples $\langle *a, b \rangle$ and $\langle *b, c \rangle$, while in the more traditional explicit representation, this alias graph is represented by $\langle *a, b \rangle$, $\langle **a, c \rangle$, $\langle *b, c \rangle$, and $\langle **a, *b \rangle$ ¹.

1. The relationship $\langle **a, c \rangle$ and $\langle **a, *b \rangle$ can be inferred from $\langle *a, b \rangle$ and $\langle *b, c \rangle$ of the points-to representation

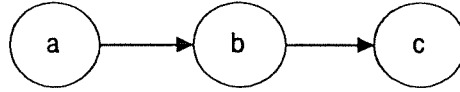


Figure 13. Example Alias Graph

Complexity Variables: Listed below are the variables that used throughout the complexity analysis discussion.

1. $NL(p)$ is the set of non-local variables that are visible in function p . In a C-like language, $NL(p)$ is the set of global variables.
2. $LOCAL(p)$ is the set of local variables of function p .
3. $LOCAL_{av}$ is the number of average local variables of all functions.
4. $PARAM(p)$ is the set of formal parameters of function p .
5. $PARAM(p, i)$ is the i th formal parameter of function p .
6. $ARGU(q, p)$ is the set of arguments of call site q that calls p .
7. $ARGU(q, i, p)$ is the i th argument of function call site q that calls p .
8. $PARGU(q, p)$ is the set of pointer arguments of call site q that calls p .
9. $PARGU_{av}$ is the average number of pointer arguments of all function calls.
10. $ALIAS(a, d)$ is the set of aliases of object a after d level of dereference.
11. AR is the number of alias relations currently holding.
12. AR_{max} is the maximum number of alias relations holding at any time.
13. AR_{av} is the average number of alias relations holding throughout execution of the program.
14. F is the number of functions in the program.
15. S is the number of pointer assignment statements in a function.
16. FC is the number of function calls in the program. In the presence of function pointers, FC can be larger than the number of physical call sites in the program.

7.3 Intra-procedural Analysis

Determining the range of possible aliases in a program is essential to decipher the behavior of the program statically—where and how a variable might be accessed carries information about the algorithm the program employs, and therefore is important to intelligent tampering or impersonation attacks.

This section examines the complexity of intra-procedural alias analysis in the presence of the flatten-n-jump technique described in section 4. Intra-procedural analysis requires the examination of the statements within a function¹ and the subsequent combinatorial analysis (if any) over their effect on aliasing. To be more specific, the intra-procedural phase entails the following operations:

- The processing of each pointer assignment statement—the effect of this particular statement has on the alias relations.
- Analysis of the combinatorial effects of all the statements (either in a flow-sensitive or insensitive manner)

The intra-procedural phase of the analysis may be repeated multiple times, for the inter-procedural analysis may result in new information that need to be processed before the alias set converges. The discussion in this section is initially focused on intra-procedural analysis only, assuming the information propagated from other functions remain static.

7.3.1 Processing of pointer assignment statements

The only kind of statements where alias information might be modified is pointer assignments of the form:

$$Ptr = Expression(Q)$$

where the evaluation of $Expression(Q)$ returns an address of a memory location, which is subsequently placed in the pointer object Ptr . Assuming p and p' are the program points immediately before and after the pointer assignment statement, and AR the set of alias relations holding at p , processing the assignment statement is to produce the set of alias relations AR' holding at p' , given the effect of executing the statement (see Figure 14).

There exists a set of well-known transfer functions f that handles different statements, some are more complex than others [MUCH97]. These transfer functions specify a set of rules via which alias relations are modified. For example, the transfer function for the statement $p = q$ where p and q are both pointers is such that,

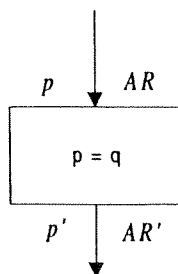


Figure 14. Processing of a pointer assignment statement “ $p = q$ ”

1. This will be done in the order of execution if flow-sensitive algorithms are used.

$$AR' = AR - (\text{any alias of } *p) + (*p, \text{deref}(q, 1))$$

This transfer function kills any alias of $*p$ (since p is reassigned) and adds the alias of $*p$ and anything q points to (since p is now pointing to whatever q points to). In Figure 14, if $AR = \{ \langle *p, a \rangle \langle *p, b \rangle \langle *q, c \rangle \}$, after the statement $p = q$, AR' is $\{ \langle *p, c \rangle \langle *q, c \rangle \}$, the alias relations $\langle *p, a \rangle \langle *p, b \rangle$ are killed by the statement.

More generally, for a statement of the form

$$Pi = Qj \tag{1}$$

where Pi denotes a pointer object i levels of dereference away from P , and Qj is a pointer object j levels of dereference away from object Q , the transfer function is described in Figure 15.

```

Transfer Function f {
  if (deref(p,i) must alias to pointer object c)
    AR' = AR - (any alias relation of c) + (*c, deref(q, j+1))
  else
    AR' = AR + { <*a, b> | a is in deref(p, I)
                  and b is in deref(q, j+1) }

```

Figure 15. Transfer Function for Statement 1

The $\text{deref}(p, i)$ function returns the set of objects that are i levels away from object P . For example, for the alias graph in Figure 13, $\text{deref}(a, 2)$ will return object c . The algorithm of $\text{deref}(p, i)$ is based on the points-to representation, and it is described in Figure 16.

```

AliasSolution <-- { }
function deref (p, derefLevel) {
  if (derefLevel = 0)
    AliasSolution <-- AliasSolution + p;
  else
    for each alias relation (*p, target) or (p, target), do
      deref (target, derefLevel -1)
    end do
  end if }

```

Figure 16. Algorithm for dereferencing pointer variables

This algorithm is similar to the alias query algorithm presented in Hind et al. [HIND99]. The average and worst case time complexity of such an algorithm is as follows, respectively,

$$O(ARsg_{av}^{\text{derefLevel}}), \text{ and}$$

$$O (ARsg_{max}^{derefLevel})$$

where $ARsg_{av}$ and $ARsg_{max}$ denote the average and the maximum number of alias relations for an object due to a single level of dereference. Most pointer assignment statements can be decomposed into statements of the form shown in (1), for which two calls to *deref* are made. The result from calling *deref* is an alias set whose size is bound by $ARmul$, and paring the results of calling two *deref* is

$$O(ARmul * ARmul).$$

Therefore the complexity of applying the transfer function to process a pointer assignment statement is roughly,

$$O (ARsg_{max}^{derefLevel} + ARmul_{max} * ARmul_{max}),$$

for the worst case, and

$$O (ARsg_{av}^{derefLevel} + ARmul_{av} * ARmul_{av}),$$

for the average case.

7.3.2 Combinatorial Analysis

In a flow-insensitive algorithm, the effect of individual statements on aliasing is simply summarized together to discern the effect of a function on aliasing. Using such an algorithm, the worst case complexity of one intra-procedural analysis phase is,

$$O (S_{max} * F * ARsg_{max}^{derefLevel} + ARmul_{max} * ARmul_{max}),$$

where S_{max} denotes the maximum number of pointer assignment statements in a function, and F denotes the number of functions in the program. In a flow-sensitive analysis, the program control-flow is taken into account in the analysis. One of the most significant differences between a flow-sensitive analysis versus an insensitive one is the treatment of the combinatorial effect of individual statements on aliasing. The flow-insensitive analysis summarizes over all statements, regardless of the order of execution. A flow-sensitive analysis, however, combines alias relations at such program points called *meet nodes* [HIND98, CHOI91]. Meet nodes are actual or abstract nodes in the flow graph where different flows meet. For example, in Figure 17, node c is a meet node where the yes and the no branch of the if statement meet.

At each meet node, a union operation on the alias relations is performed, and the complexity of that operation is bound by, again, the maximum number of alias relations holding at any time during execution. This union operation is

$$O(\text{number of meet nodes} * ARmax)$$

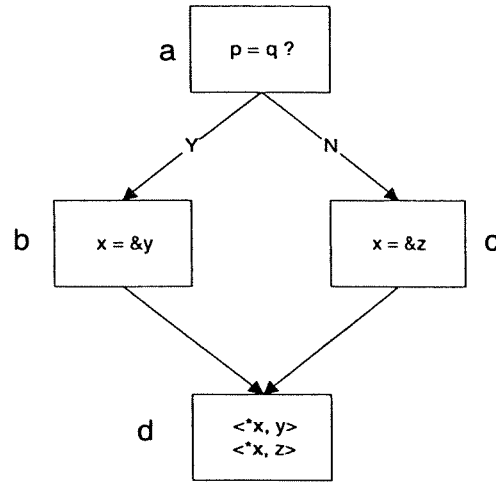


Figure 17. Example of a Meet Node

If the flow graph is fully flattened, there exists only one meet node in each function—the switch statement node (see Figure 5). The complexity of this analysis step is therefore straightforward:

$$O(F * ARmax),$$

where F is the number of functions and $ARmax$ denotes the largest number of alias relations holding at any execution point. Thus, the complexity of one flow-sensitive intra-procedural analysis phase can be described as,

$$O(S_{max} * F * ARsg_{max}^{derefLevel} + ARmul_{max} * ARmul_{max} + F * ARmax),$$

7.4 Inter-procedural Alias Analysis

As described in section 4.4, inter-procedural alias analysis primarily handles information propagation between the calling and the called function. In particular, two information propagation processes are needed in order to discern the effect of function calls. They are: *forward binding* and *backward binding*.

Forward binding maps the set of aliases holding at a call site in function f into aliases holding on entry into the called function g , using the argument/parameter mappings of the function invocation. Backward binding maps aliases holding at the exit of the invoked function g into aliases holding immediately following the execution of g in f .

This section investigates the complexity of the forward and backward binding process, for they directly affects the efficacy of the inter-procedural code transforms. The complexity discussion continues in the context of a C-like language with pass-by-value parameters and pointers.

For every function call, the forward binding is shown in Figure 18. This forward binding algo-

```

ForwardBinding( ) {
  for each pointer variable a in ARGU(q, p) such that
    a is ARGU(q, i, p), do
    replace <*a, x> (or <a, x>) in AR with <*PARAM(p, i), x>
    (or <PARAM(p, i), x>)
  end do
end forwardBinding}

```

Figure 18. Forward Binding Algorithm

rithm simply replaces all occurrences of an actual pointer parameter in the alias relation with the corresponding formal parameter. For example, performing the forward binding process for the function call in Figure 19 would result in $\{<*m, a>, <*n, a>\}$ holding at the entry of function g .

The forwarding binding algorithm in Figure 18 is devised based on the points-to alias abstraction, and it is otherwise general in the sense that the complexity of the algorithm does not depend on any implementation of data structures specific to the algorithm. The average and worst case time complexity for the forward-binding process are such as follows:

$$O(PAGU(q, p) * AR_{av}), \quad \text{and,}$$

$$O(PARGU(q, p), AR_{max})$$

where $PARGU(q, p)$ is the set of pointer arguments at call site q that calls p , and AR_{av}, AR_{max} denote the average and the maximum number of alias relations holding during execution.

The backward binding process, in the presence of pass-by-value and pointer arguments, is described in Figure 20.

The backward binding process simply discards each alias relation that involves a local variable in the called function, and replaces alias relations of the form $<*PARAM(p, j), x>$ with $<*ARGU(q, i$

```

int a,b;
f() {
  int *x, *y;
  x = &a;
  y = &a;
  -----<*x, a> <*y, a> hold at this point
  g(x, y);
}

g(int *m, int *n) {
  int *x;
  x = &b;
  n = x;
  *m = *x + 1;
}

```

Figure 19. Example Illustrating Forward Binding

```

BackwardBinding ( ) {
  for each(x, y) in current alias relation AR, do
    if either x or y is in LOCAL(p),
      discard (x, y) from AR
    else
      replace (x, y) where x is PARAM(p, i) with (ARGU(q, i, p), y)
    end if
  end do
}

```

Figure 20. Backward Binding Algorithm

p), x>. Time complexity of the backward binding process is, $O(AR_{av} * LOCAL(p))$, or $O(AR_{max} * LOCAL(p))$ for the worst case. Consider again the example in Figure 19. The set of alias relations holding at the exit of function g is $\{ \langle *m, a \rangle, \langle *n, b \rangle, \langle *x, b \rangle \}$. Performing the backward-binding algorithm would result in elimination of the alias relation $\langle *x, b \rangle$ since x is only local to g. The alias relations holding at the return from g are, $\{ \langle *x, a \rangle, \langle *y, b \rangle \}$. For each function call, the cost of propagating alias information back and forth between the called and calling function is,

$$O(AR_{av} * (LOCAL(p) + PARGU(q, p))), \quad \text{and}$$

$$O(AR_{max} * (LOCAL(p) + PARGU(q, p))), \quad \text{for the worst case.}$$

Thus the overall time complexity in handling inter-procedural information propagation is:

$$O(FC * AR_{av} * (LOCAL_{av} + PARGU_{av})) \quad (2)$$

where FC is the number of edges in the PCG.

The inter-procedural binding process is further complicated by the fact that function calls can happen recursively. Whenever recursive function calls are involved, the backward binding process needs to be more conservative in order not to result in incorrect analysis outcome. For example, when a calling function is also visible from a called function due to recursion, the alias relations concerning only local variables of the called function may carry into the calling function. Therefore, the backward binding process must distinguish which local variables might be visible to the calling function (perhaps passed with an "&" operator), and care must be taken that these alias relations do not get discarded upon exit from the called function.

7.5 Iterations Over the PCG

Described in section 7.3 and 7.4 are complexity measures for intra and inter-procedural alias analysis. The complexity formulas (1) and (2) represent one-time cost for conducting intra and inter-procedural analysis. However, an aggressive analysis algorithm might attempt to iterate over the PCG and repeat the process of (1) and (2) multiple times before the alias set converges. Using such an iterative algorithm, the maximum number of iterations over the PCG is

$$O(AR_{max} * F),$$

where AR_{max} is the maximum number of alias relations holding at any time and F is the number of functions. During each iteration, the intra-procedural and the inter-procedural phase might be repeated; that is, the overall complexity of iterating over PCG and repeating the intra and inter-procedural analysis is,

$$O((AR_{max} * F) * ((FC * AR_{max} * (LOCAL_{max} + PARGU_{max})) + (S_{max} * F * AR_{sg_{max}}^{derefLevel} + AR_{mul_{max}} * AR_{mul_{max}} + F * AR_{max}))) \quad (3)$$

7.6 Putting Together the Complexity Argument

The complexity measures represent the worst-case time requirement for a full-up alias analysis. As shown in (3), what is of consequence is the size of the alias set AR , the size of the parameter set ($PARGU$ and $LOCAL$), and the size of the program (S and F).

Worst-case complexity, however, is often not an accurate indicator of performance. In practice, what is of interest is the number of times a function is visited and the number of times a transfer function is evaluated during analysis. For a program with a flattened control flow, a degenerate call structure and an abundant number of aliases, a function can be visited each time a function call site is encountered (consider a fully connected program call graph), and a basic block will be analyzed each time a branch instruction is met (consider a fully connected flow graph). In such a case, the number of times a particular function is visited and a pointer assignment statement is analyzed is not far from the worst case estimate. Such an analysis will eventually converge and report a set of results that include as many spurious alias relations as it possibly can: in several occasions when tested against automated analysis algorithms, the algorithm halts with a report that every pointer variable is aliased to every memory location that appears on the right hand side of some assignment statement.

We have finally reached the major result of the transformations that we have performed on programs:

The degeneration of the program control flow and call structure renders a data-dependent program flow structure. That is, control-flow and data-flow analysis are made co-dependent.

The result of this co-dependence are: (1) vastly increased complexity for both control-flow and data-flow analyses, and (2) reduced analysis precision. The practical result is that automatic static analysis of a transformed program is essentially impossible.

8 PERFORMANCE RESULTS AND EMPIRICAL EVALUATION

In this section we report performance results obtained with experimental transformations on the SPEC95 benchmark programs. Of issue here are three measures: *performance of the transformed program*, *performance of static analysis*, and *precision of static analysis*.

By performance of the transformed programs, we mean the execution time and the executable object size after transformation. These measures reflect the cost of the transformation. By performance of static analysis, we mean the time taken for the analysis tool to reach closure and terminate. A related but equally important criterion is the precision of static analysis, which indicates how accurate the analysis result is compared to the true alias relationships.

8.1 Performance of the Transformed Program

The following data was obtained by applying our transforms to SPEC95 benchmark programs. Three SPEC programs were used in this experiment, *Compress95*, *Go* and *LI*. *Go* is a branch-intensive implementation of the Chinese board game GO. *Compress95* implements a tightly-looping compress algorithm, and *LI* is a LISP interpreter program. These programs embody a wide range of high-level language constructs, and therefore are good representatives of real-world programs.

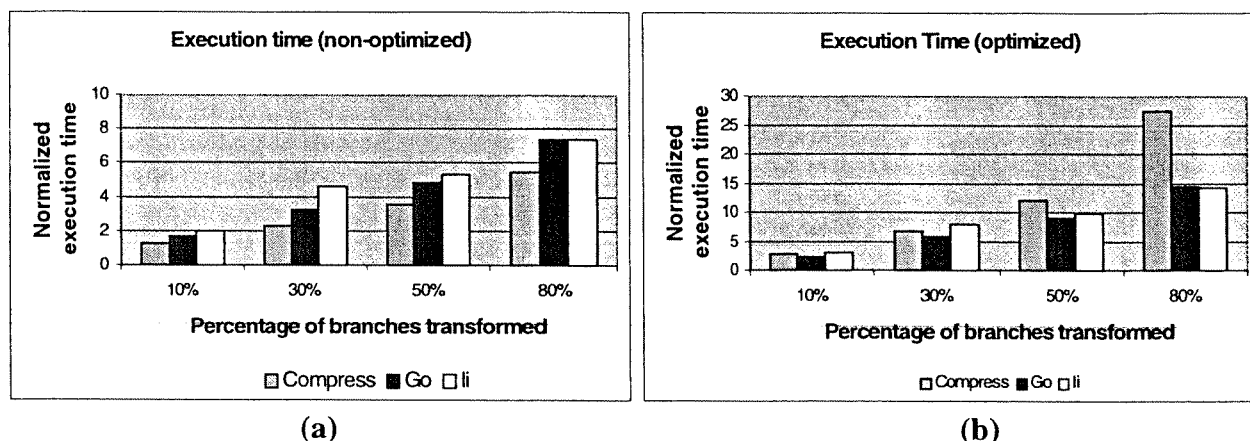


Figure 21. Execution time of the transformed benchmark programs

We conducted experiments on both optimized (with the gcc -O option) and non-optimized versions of the programs. The experiments were executed on a SPARC server. Results from the experiments are depicted in Figure 21.

As the performance data show, in both optimized and non-optimized cases, the performance-slowdown increases exponentially with the percentage of transformed branches in the program. On average, performance-slowdown is significantly worse in the optimized case. This is encouraging because what we observed was that our transformation considerably hindered the optimization that the compiler is able to perform.

The performance of Go and LI were similar for both optimized and non-optimized code. Of all three untransformed programs, compiler optimization performed best on *Compress95*—a whopping 80% decrease in the execution time due to optimization. However, as can be seen in Figure 21(a), our transforms removed significant optimization potential from *Compress95*; the execution speed of the transformed and optimized *Compress95* diverges most significantly from that of the original optimized program. As *Compress95* is a loop-intensive program, it is likely that certain analysis which enabled significant loop or loop kernel optimization was no longer possible after

our transform was performed.

The executable object size of the three benchmarks grew with increased branch replacement. The experimental data of program size expansion is shown in Figure 22. Go, a branch-intensive program, displays the largest code growth with our transform. For 80% replacement of direct branches, the executable size increased by a factor of 3 for Go and LI, and by roughly 10% for Compress95. Compress95 contains relatively fewer static branches which resulted in less code growth with the transforms.

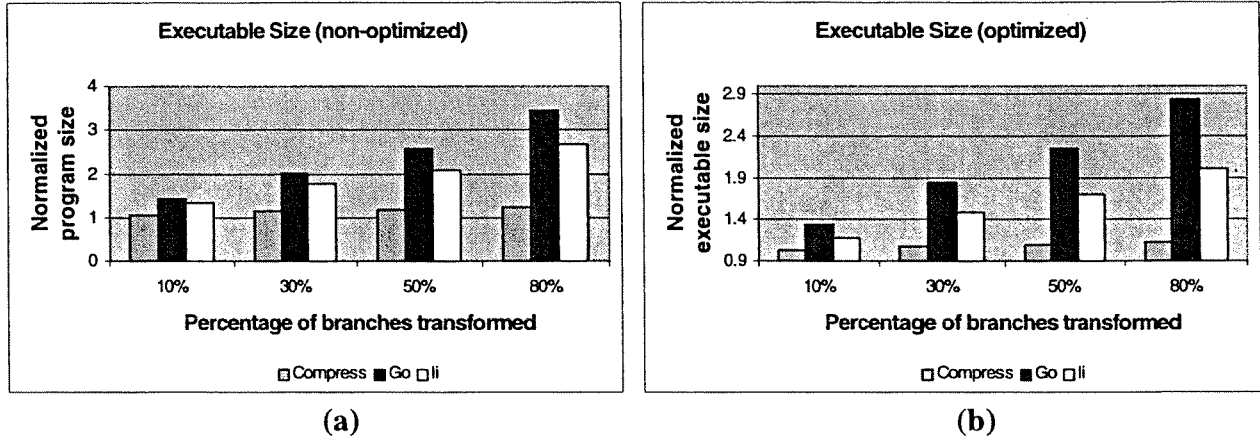


Figure 22. Program size of the transformed benchmark programs

We believe that these results are representative of many programs. It appears that replacing 50% of the branches will result in an increase of a factor of 4 in the execution speed of the program. At the same time, the program will almost double in size. In these experiments, we used a random algorithm to choose which branch to transform. An obvious improvement is to employ intelligence to do the following: a) identify the regions of the program that require greater protection from static analysis, and b) selectively perform transformation on the less-often-executed branches for better performance penalty. Trade-offs between these two criteria need to be considered for the most effective solution.

8.2 Performance and Precision of Static Analysis

In this experiment, we report the result of running existing static analysis tools. The notable static analysis tools include the NPIC tool from IBM [HIND99] and the PAF toolkit from Rutgers university [PAF]. The NPIC tool implements a sophisticated algorithm, and represents the state-of-the-art in the field of static analysis. Unfortunately, IBM no longer maintains and distributes the tool. Our empirical results therefore are obtained using the PAF toolkit. PAF implements a flow-sensitive, inter-procedural pointer analysis algorithm [LARY92]. Both NPIC and PAF perform control-flow analysis exactly once with no further refinement on the flow graph.

In our experiments, *before* transformation PAF successfully analyzed small, toy programs but failed to handle some of the large programs included in the SPEC benchmarks. We tested PAF on a wide range of small programs that contain extensive looping constructs and branching state-

ments. In all of the test cases, PAF terminated reporting the largest possible number of aliases in the program (the worst possible precision)—in a program with n distinct pointer assignments and k basic blocks, it reports $n*k$ alias relations. Because of the size of the test programs, we observed negligible differences in the pre- and post-transformation analysis time. The experience with the PAF tool, albeit with limited test cases, indicated that PAF failed to resolve aliases across the flattened basic blocks, and that our technique of making data-flow and control-flow co-dependent presents a fundamental difficulty that existing analysis algorithms lack the sophistication to handle.

9 IMPLEMENTATION

In this section we briefly discuss the implementation of a prototype of the One-way Translation mechanism. We identified three design goals for the One-way Translation compiler. They are:

- Platform-independence
- Modular support for code transforms
- Easy extension to other high-level languages

To meet these design goals, we selected ANSI C as the target language and the SUIF compiler from Stanford as the infrastructure to implement code transformations[SUIF].

The SUIF compiler toolkit offers an extensive set of utilities for source-code manipulation. The current SUIF system uses an intermediate representation called SUIF and a set of well-defined functionality to manipulate the SUIF representation. It takes as input a C program, performs code transformations, and produces as output a new ANSI C program. The resulting C code can then be compiled using any ANSI C compiler. Since the code transforms operate on the SUIF representation which is largely architecture independent, these choices meet the first requirement, platform-independence.

The types of code transformation may evolve as new protection techniques surface. It is therefore important to have modular support for incorporating additional transformations. In SUIF, code transformations are implemented as SUIF passes, and SUIF provides a programming environment that allows the independent development and easy integration and inter-operation of different passes. This made SUIF an ideal implementation environment for the One-way Translation compiler.

Finally, source-to-source translation tools exist between ANSI C and several other languages such as C++, Pascal and Fortran. Programs written in those languages can be translated into ANSI C before undergoing the code transforms. Therefore the choice of ANSI C as the target language allows easy extension to other languages. The SUIF team is currently designing a JAVA front-end, which when available, can be incorporated into our One-way Translation compiler to handle JAVA programs.

At this level of automation, the programmer is required to specify transformation parameters such as the level of degeneration and aliasing. Fortunately, this step is not terribly burdensome since it involves no more than setting the values of a few parameters.

10 CORRECTNESS DISCUSSION

When programs are transformed during compilation, there is always the issue of whether the transformation is performed correctly—whether the resulting code preserves the semantics of the original program.

Proving the correctness of the transforms formally is beyond the immediate scope of this work. In fact, the compiler community has not solved the general correctness question regarding the more traditional code transformations performed by optimizing compilers. While the correctness issue is not dealt with directly in this work, we point to a few research ideas that address the various aspects of the correctness problem.

Translation validation [PSS98] is a technique designed to check the result of each compilation against the source program and pinpoint errors on-the-fly. Nacula proposed a practical framework for translation validation within which small instances of code transforms (described as a pass) can be validated [NECU00]. Nacula showed that one can implement a practical translation validation infrastructure with about the effort typically required to implement one compiler pass. Since our transformations are implemented as compiler passes, we believe that while we may not be able to prove that our compiler is always correct, we can at least check the correctness of each compilation using translation validation techniques.

11 DEFENDING AGAINST DYNAMIC ANALYSIS

Dynamic analysis of programs include execution simulation, profiling and debugging. At the writing of this document, the mechanisms to defend against dynamic analysis are very much work-in-progress, and they are documented below.

Our One-way Translation technique applies randomizations to the program that are designed specifically to induce analysis complexity. The randomness introduced in the program can defend against dynamic as well as static analyses. The following paragraphs explain why this is the case.

Dynamic analysis of an executing program can be viewed as a special case of software testing. The general problem of comprehensive program testing is known to be difficult [ABK88], for the number of potential paths through the program execution can be exponential in the size of the program. Comprehensive testing, therefore, faces the issue of completeness—whether the test cases cover all possible paths.

If we take this observation a step further and consider it in the context of software protection, dynamic analysis of programs can be effectively thwarted if the following are true:

- Dynamic analysis of program for security purposes is as difficult as conducting a comprehensive path testing for the program.
- The number of potential paths through the program is made intentionally large such that testing the program will be essentially difficult.

Another potentially fruitful idea is to explore non-determinism in programs. Multi-threaded programs, for example, exhibit non-deterministic behavior, which is problematic for program simula-

tion. If the concurrent threads execute in a random order, and there are a large number of possible execution scenarios due to concurrency, the chance of simulating the program and actually learning something definitive and useful about the current execution is fairly small.

Non-determinism is a powerful and promising technique which can change the control flow of the program in a substantial way—the reverse mapping from the concurrent binary threads back to the sequential control flow of the source code will be extremely difficult. However, it is not a trivial task to convert sequential programs that are not originally designed for concurrency into concurrent programs. We will investigate the feasibility of such an approach in pursuing two possible directions—restructuring the program into a set of cooperating tasks and adding benign tasks that provide minimal yet useful service. We suspect that some hybrid of the two will have to be adopted.

All these ideas are rooted in the same principle of state inflation. That is, the internal state space of the program is made intentionally large such that any realistic analysis cannot be completed within reasonable amount of time. This can be explained using a simple state machine model of programs.

If we view programs as state machines, state transitions are prompted by program execution. Each instruction executed emits some information into its environment that can be gathered and used in program analysis. Most real world programs have a limited number of states, and program execution typically revisits these states (e.g. in a loop). If an adversary is able to discern that a previously visited state is entered again, they will know how execution will proceed after that (simply repeating the set of states since the last visit). Inflating the state space does not imply a blind addition of states. However, it involves a careful reorganization of the state space such that the first occurrence of a recognizable state does not occur within a prescribed time frame. A potential complication is the performance issue once the state space is increased. Care should be taken so that it does not seriously hinder the performance of the program.

Dynamic profiling and debugging is yet another technique for program analysis. For example, once a control-flow graph (CFG) is built, a clever trick is to group the basic blocks and edges of the CFG into equivalence classes based on frequency of execution determined by the profiling analysis. It is then trivial to map one CFG onto another, assuming the two graphs do correspond to each other.

This type of analysis can be thwarted by adopting techniques that confuse the profiling algorithm deliberately. For example, benign code that does non-critical but useful computations can be added to the program to muddle with the execution count obtained by sampling of the program counters. In addition, spurious computations on consequential variables can be used to confuse the profiling process.

12 RELATED WORK

Protecting trusted software from untrustworthy hosts has many potential applications. The copyright protection industry has long employed some of the methods presented in this document. Another context in which this problem is of interest is the mobile-code environment [FGS96, MEAD97, YEE97]. Mobile programs traverse from machine to machine, and in some cases it is

necessary to ensure that the integrity of the program is preserved by the chain of executing hosts.

A few studies have been reported that were concerned with the protection of mobile agents from malicious hosts. Most of the work is done to protect Java byte code [CTL97a, CTL97b], and these studies have typically concentrated on code-obfuscation techniques. While these techniques are innovative in their own right, they lack the basis of formal analysis and provable results. Most of these techniques are based on loosely-defined software complexity metrics that bear no real correspondence to program understandability. For example, one of the complexity metrics used in evaluating how well the code has been obfuscated considers that two-dimensional arrays are more complex than arrays of only one dimension [MUNS93]. However, this overlooks the fact that in some cases, it is more intuitive and simpler to represent the data in a multi-dimensional array than otherwise. These techniques also assume that the attacker does not have access to the source code while we assume a very sophisticated adversary who knows all there is to know about the source code before the transformations. These obfuscation operations occur only at the source code level while we employ randomization and obfuscation techniques at all stages of the code generation – from source code to run-time.

One particularly relevant area is that of mobile cryptography [SATS98]. This work raises the possibility that programs could be executed in an encrypted form. The approach is able to provide code secrecy and integrity automatically, but the technique, as it stands now, only applies to polynomial and rational functions. It remains to be seen if it can be extended to work with general-purpose programs.

Our problem context is fundamentally different from that of a mobile code environment. First of all, if a mobile program fails to survive, it will not in general lead to system-wide damage. Second, mobile code may execute on any arbitrary platform that it might or might not know anything about. In that sense, building survivable mobile code is more difficult than the problem we set out to solve. In our problem context, we have some control over the host machines, although we have to allow the possibility that these machines can be compromised. This implies that we can make more precise assumptions about the executing environment. Our one-way translation techniques can be extended to the application software executing on the host machine if necessary, while mobile agents will not be able to change anything on the host machine. We believe that viable techniques can be devised to build secure survivability schemes, while solving the corresponding problem in the mobile environment may be an unattainable goal.

Aucsmith's work on the Integrity Verification Kernel (IVK) [AUCS96] at Intel is of direct interest because it puts forth the concept of building tamper-resistant software. An IVK consists of multiple cells (code segments), and all the cells are encrypted except the one that is currently executing. The executed cell becomes encrypted again after execution while the next cell decrypts. This method requires significant computational resources to accomplish the dynamic encryption and decryption process. In addition, cumbersome manual intervention on the part of the programmer is needed in order to identify critical code segments that must be specially armored to create IVKs in the first place.

The Immunix project at Oregon Graduate Institute includes an effort to build survivable operating systems through diversity specialization [PBCW96]. The concern in that work is mainly to thwart class attacks due to software flaws. The claim is that the specialization method allows the system

to guard the validity of the operating-system software, both statically and dynamically. It is unclear, and the designers of Immunix provided little hint, whether diversity techniques can be as effective for complex software such as operating systems. Immunix's stack guard work is also of interest to us, but of a slightly different kind. Instead of placing canary words in the stack frame to detect buffer overrun attacks, our method would call for a randomization in the placement of return addresses to prevent buffer overrun attacks from happening in the first place.

Another approach worth noting is Devanbu and Stubblebine's work on protecting stack and queue integrity on hostile platforms [DEST98]. Their method uses digital signature chains, starting from an initial signature that is protected on a trusted device, to verify the integrity of the data and data operations. A trusted device handles the signature computation and data generation, while the hostile host manages the data structure. This work does not address methods for protecting general-purpose software.

13 SUMMARY

Motivation of this work initially stems from the important issue of protecting network survivability mechanisms. The proper protection of such mechanisms is essential but difficult because parts of the mechanism may operate on the untrustworthy application hosts.

Our solution to this problem is to inhibit program analysis by a suite of mechanisms including One-way Translation and diversity schemes. The translation is made one way by embedding randomness in the translation process. Our approach precludes program analysis under many circumstances, or at least makes it an extremely expensive operation.

One-way translation is not a perfect security mechanism. There are scenarios in which an adversary could defeat the protection provided but we claim that they are the most unlikely to occur because they require extreme system access. If such access were possible (because of insider access) then there are many simpler ways for the adversary to proceed. One-way translation is a compromise based on maximizing the protection provided for what is expected to be reasonable cost.

14 REFERENCES

- [AETA97] Anderson *et al.* *Continuous Profiling: Where Have All the Cycles Gone?* ACM Transactions on Computer Systems, November 1997, Vol 15, No.4.
- [ABK88] P. Amman, S.S. Brilliant, and J.C. Knight, *The Effect Of Imperfect Error Detection On Reliability Assessment Via Life Testing*, IEEE Transactions on Software Engineering Vol. 20, No. 2, February 1994.
- [AUCS96] D. Aucsmith, *Tamper Resistant Software*. Proceedings of the First Information Hiding Workshop. Cambridge, UK
- [BANN79] J. P. Banning. *An Efficient way to find the side effects of procedure calls and the aliases of variables*. In Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages. pp29-41, January 1979.
- [BLAK79] G. R. Blakley. *Safeguardin Cryptographic Keys*. Proceedings of the National Com-

- puter Conference, 1979. Vol 48.
- [CER99] C. Cifeuentes, M. Van. Emmerik, N. Ramsey. The Design of a Resourceable and Retargetable Binary Translator. Proceedings of the Sixth Working Conference on Reverse Engineering, Atlanta, USA. October, 1999. pp280-291.
 - [CHOI91] J. Choi, R. Cytron, J. Ferrante. *Automatic Construction of Sparse Data Flow Evaluation Graphs*. In 18th Annual ACM Symposium on the Principles of Programming Languages. pp55-66.
 - [CTL97a] C. Collberg, C. Thomborson, D. Low. *Breaking abstractions and unstructuring data structures*, IEEE International Conference on Computer Languages, Chicago, May 1998.
 - [CTL97b] C. Collberg, C. Thomborson, and D. Low. *A taxonomy of obfuscating transformations*. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.
 - [DEST98] P. Devanbu and S. Stubblebine. *Stack and Queue integrity on hostile platforms*. Proceedings of IEEE Symposium on Security and Privacy, Oakland, California, May 1998.
 - [DSB96] Report of the Defense Science Board Task Force On Information Warfare -- Defense (IW-D), Office of the Secretary of Defense. November 1996. Available at <http://www.jya.com/iwd.htm>
 - [ELKN99] This is the updated version of Matt's tech report.
 - [EMAM94] M. Emami, R. Ghiya, L. Hendren. *Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers*, In Proceeding of of the SIGPLAN 94 Conference on Programming Language Design and Implementation, pp242-256. SIGPLAN Notices, Vol 29. No. 6. Orlando, Florida USA.
 - [FGS96] W. Farmer, J. D. Guttman, and V. Swarup. Security for Mobile Agents: Issues and Requirements. In Proceedings of the 19th National Information System Security Conference, Baltimore, Maryland.
 - [FERR98] A. Ferrari. *Process State Capture and Recovery in High-Performance Heterogeneous Distributed Computing Systems*. Ph.D. Dissertation. University of Virginia, January, 1998.
 - [FOSO96] S. Forrest, A. Soma. *Building Diverse Computer Systems*. In the Proceeding of the 1996 Hot Topics of Operating Systems Conference.
 - [GRID97] S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, S. Staniford-Chen, R. Yip, D. Zerkle. *GrIDS: A Graph-Based Intrusion Detection System*. In the Proceeding of the 1997 National Information System and Security Conference, Baltimore, 1997.
 - [HECH77] M. Hecht. *Flow Analysis of Computer Programs*, Elsevier North-Holland Press, 1977.
 - [HIND98] M. Hind, A. Pioli, *Assessing the Effects of flow-sensitivity on Pointer Alias Analyses*. Research Report 21251, IBM T. J. Watson Research Center.
 - [HIND99] M. Hind, M. Burke, P. Carini and J. Choi. *Inter-procedural Pointer Analysis*. ACM Transactions on Programming Languages and Systems, Vol. 21, No. 4, July 1999, pp 848-894.
 - [IDIP97] D. Schnackenberg. *IDIP concept Document*. Boeing, Personal Communication

- [INGE71] F. M. Ingels. *Information and Coding Theory*. Intext Educational Publisher, 1971.
- [KEHO97] R. Keller, U. Holzle. *Binary Component Adaptation*. Technical Report, Department of Computer Science, University of California at Santa Barbara, TRCS-97-20.
- [KEP97] J. Knight, M. Elder, J. Flynn, P. Summary of *three critical infrastructure systems*. Computer Science Technical Report, CS-97-27, Department of Computer Science, University of Virginia.
- [LAMB73] H. Lambert, *System Safety Analysis and Fault Tree Analysis*, Lawrence Livermore Laboratory Report, UCID-16238, 1973
- [LAN921] W. Landi. *Undecidability of Static Analysis*. ACM Letters on Programming Languages and Systems, Vol. 1, No. 4. December 1992, pp 323-337.
- [LAN922] W. Landi. *Interprocedural Aliasing in the Presence of Pointers*. Ph.D. Dissertation, Rutgers University, 1992.
- [LARY92] W. Landi, B. Ryder. *A Safe Approximate Algorithm for Interprocedural Pointer Aliasing*. In Proceedings of the 1992 SIGPLAN Symposium on Programming Language Design and Implementation. pp235-248, June 1992.
- [LUNT93] T. Lunt. *Detecting Intruders in Computer Systems*. 1993 Conference on Auditing and Computer Technology. 1993.
- [MARL93] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. D. Choi, M. G. Burke, and P. Carini. *Pointer-induced Aliasing: A clarification*. ACM SIGPLAN Notices, Vol 28, No. 9. pp67-70, September 1993
- [MEAD97] C. Meadows. *Detecting attacks on mobile agents*. In Proceedings of the DARPA workshop on Foundations for secure mobile code, Monterey CA, USA, March 1997.
- [MUCH97] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
- [MUNS93] J. C. Munson and T. M. Kohshgoftaar. *Measurement of data structure complexity*. Journal of Systems software, 20:217-225, 1993.
- [MYER81] E. Myers. *A Precise Inter-procedural Data Flow Algorithm*. In the conference record of the Eighth Annual ACM Symposium on Principles of Programming Languages. Williamsburg, VA. January, 1981. pp219-230
- [NECU00] G. Necula. *Translation Validation for an Optimizing Compiler*. In the proceeding of the ACM SIGPLAN Conference on Programming Language Design and Implementation. May 2000. Vancouver, British Columbia, Canada.
- [NEUM00] P. Neumann, *Practical Architectures for Survivable Systems*, Report for the Army Research Lab. 2000.
- [PAF] *The Prolangs Analysis Framework*. Rutgers University. <http://www.prolangs.rutgers.edu/>
- [PBCW96] C. Pu, A. Black, C. Cowan, J. Walpole, *A Specialization Toolkit to Increase the Diversity in Operating Systems*. Proceedings: 1996 ICMAS Workshop on Immunity-based systems. Nara, Japan. December 1996.

- [PCCP97] Report of The Presidential Commission on Critical Infrastructure Protection, 1997.
- [PONE97] P. Porras, P. Neumann, EMERALD: *Event Monitoring Enabling Responses to Anomalous Live Disturbances*. In proceedings of the 1997 National Information Systems Security Conference. Baltimore, 1997.
- [PSS98] A. Pnueli, M. Siegel, and El. Singerman. *Translation Validation*. In Bernhard Steffen, editor, Tools and Algorithms for Construction and Analysis of Systems. 4th International Conference, TACAS'98, volume LNCS1384, pp151-166. Springer 1998.
- [RISK94] P. Neumann, *Computer-Related Risks*. ACM Press, New York, and Addison-Wesley, Reading, MA, 1994.
- [RSA78] R. Rivest, A. Shamir, Adleman. *A method for Obtaining Digital Signatures and Public-Key Cryptosystems*, Communications of the ACM 21,2, February 1978, pp120--126.
- [SATS98] T. Sander, C. Tschudin. *Protecting Mobile Agents Against Malicious Hosts*. Lecture Notes of Computer Science, Vol 1419. Mobile Agents. 1998. Springer-Verlag.
- [SKDG98] K. Sullivan, J. C. Knight, X. Du, and S. Geist. *Information Survivability Control Systems*. Proceedings of the International Conference of Software Engineering, 1998.
- [SHAM79] A. Shamir. *How to share a secret?* Communication of the ACM, Vol. 22, Nov. 1979. pp 612-613.
- [SUIF] G. Aigner, et al. *The SUIF2 Compiler Infrastructure*. Documentation of the Computer Systems Laboratory, Stanford University.
- [SW63] C. E. Shannon, and W. Weaver. *The Mathematical Theory Of Communication* University of Illinois Press, Urbana and Chicago. 1963
- [THHA91] M. Theimer, and B. Hayes. *Heterogeneous Process Migration by Recompile*, Proceedings of the 11th International. Conference on Distributed Computing Systems, Arlington, TX. May 1991
- [VGRH81] W. E. Vesely, F. F. Goldverg, N. H. Roberts and D. F. Haasl, "*Fault-tree handbook*". U.S. Nuclear Regulatory Commission Rep. NUREG-0492, 1981.
- [YEE97] B. Yee. *A Sanctuary for Mobile Agents*, In Proceedings of the DARPA workshop on Foundations for secure mobile code, Monterey CA, USA, March 1997.

APPENDIX C

“Solutions for Trust of Applications on Untrustworthy Systems”

Solutions for Trust of Applications on Untrustworthy Systems

Jonathan C. Hill, Jack Davidson, John Knight
<jch8f, davidson, knight>@cs.virginia.edu

Distributed systems rely on non-local applications. At the same time, non-local applications can only be trusted as far as a non-local systems can be trusted. This is inadequate for the purpose of monitoring and maintaining critical infrastructure that relies on a distributed computer system. We require a distributed, flexible, and reliable application system to act non-locally throughout a network. Flexibility encourages a model that utilizes application level processes, dispatched from a trusted source system to untrustworthy non-local systems in the network. Reliability requires that the local system be aware of the state of operation of its dispatched application on the inherently untrustworthy non-local system. Unfortunately, these requirements lead to a scenario of a trust gap, in which a dispatched application level process must correctly function while relying on non-local (to the dispatcher, local to the application) system services which cannot be trusted. This is the inverse problem of the untrustworthy incoming application, in which a trusted system is asked to support an untrustworthy application. As such, a trust gap comes with a critical, unique, and difficult set of properties of great importance for the development of fault tolerant distributed systems. In this paper we will consider hardware and low-level software solutions to the trust gap problem. We develop a taxonomy of possible solutions and investigate the promise of each approach. Of these, we find two solution approaches with potential and applicability to today's distributed computing environments.

1.0 Introduction

Critical distributed infrastructures must react to faults; reconfiguring applications, distributing new tasks and priorities, and preparing to defend against additional faults. Without such capability, the survival potential for critical networks is highly questionable. Yet today's critical distributed infrastructures are not prepared to handle non-local faults in an automated manner. As it stands, what happens to one part of a network is either watched helplessly by the remaining network, or only adapted to locally by some simple distributed algorithm. This is regardless of the fault's impact on overall network functionality. What is needed is a network that can respond in a more intelligent manner. The ideal network would react more as organism than architecture, responding to faults with corrections, healing, and posturing, rather than passively crumbling and awaiting manual repairs upon each deterioration.

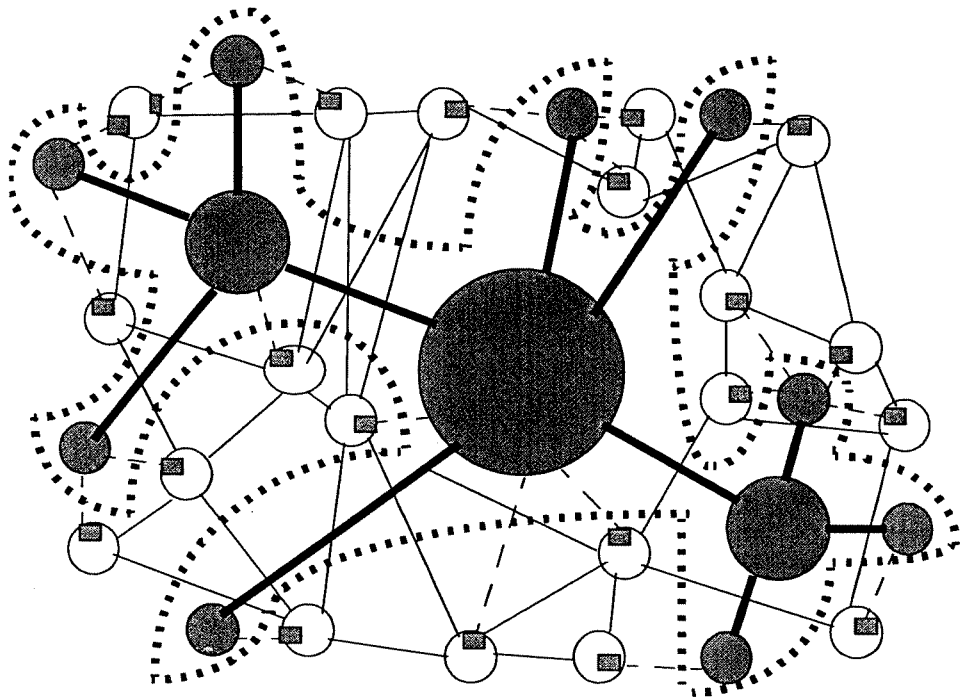


FIGURE 1.

A graph of a fault-tolerant distributed network consisting of an untrustworthy work network (clear nodes and thin connections) with a hierarchical reactive "nervous system" support network (shaded nodes with thick connections.) Work nodes are monitored by the support nodes. The dashed curved line indicates the trust boundary between the support network and the work network. The small shaded rectangles within work nodes are support system applications which must maintain clarity of their functionality while executing on the untrustworthy work nodes.

A critical infrastructure composed of a large, heterogeneous distributed system must include network wide, non-local fault tolerance in order to maintain survivability. What fails for one portion of the network must not be ignored by the rest of the network. We

propose a network architecture that contains a “nervous system”, and hence contains a highly guarded, reactive, information propagating and hierarchical control infrastructure. Such a network is depicted in figure 1.

The network depicted in figure 1 contains “work” nodes and “support” nodes. Work nodes perform the work of the distributed system, while “support” nodes are responsible for maintaining the clarity of the distributed system. The support nodes of the network are carefully maintained and highly guarded. As a result they are trusted by the network, much as a nervous system is highly guarded and trusted by an organism. Meanwhile, the work nodes are numerous and not intrinsically guarded. Thus the clarity of work nodes is assumed suspect by the network as a whole.

The support nodes act as a nervous system for the network. They report and organize conditions on the network, as well as respond to changing network conditions. Support nodes detect non-local faults in the network and react, reconfiguring the network in the presence of damage, responding preemptively to threats, and rearranging distributed services as needed. In general, it is the support node system which is responsible for the effective arrangement of the distributed system’s work and services.

For the remainder of this paper, internal trust of the nervous system is a stated assumption of the network. In other words, the nervous system, as a network, trusts itself.

Due to the trusted nature of the support nodes and the untrustworthy nature of the work nodes, a trust boundary exists at the interface to the work nodes. This trust boundary is depicted in figure 1. From the perspective of the nervous system network,

we cannot assume that the work nodes are trusted.

In particular, it is network wide non-local faults in the work nodes and their network that we intend to tolerate. It is the natural relationship between the fault-catcher (nervous system) and the fault-capable network (work network) that renders the work nodes inherently untrustworthy. Yet,

it is of critical importance for the reactive system that trustable information gathering and reconfiguration be accomplished on the untrustworthy work nodes.

If the network is to survive an attack (as an example,) then it must obtain accurate information from potentially compromised work nodes, as well as enforce actions on these nodes. As a result,

the nervous system network is required to enforce non-local action where trust cannot be given.

This is a principal problem of fault-tolerant network design. How do we maintain clarity of sensor/actuator applications dispatched by a trusted nervous system across the trust boundary to work nodes? Complicating this question are goals such as portability and reconfigurability. Our fault-tolerant network must be able to reconfigure and manage application distribution over the work network, dynamically, with maximum flexibility. Meanwhile, the work node network may consist of a large collection of COTS compo-

nents as well as legacy systems. For this reason, the application level for dispatched sensor/actuators best suits flexibility and dynamic capability.

Work at the application and high-system level has yielded interesting methods of protecting applications [Wang, Hill, Knight, Davidson 00.] These techniques harbour protection from static analysis of the application within the application itself, with the original static analysis information of the program being a secret held by the trusted nervous system. While this hinders intelligent tampering and impersonation attacks, we still require a solution that protects us from the root causes that allow protection violations and attacks in the first place. The root cause is the lack of trust provided by the underlying system on the work node. Therefore, this paper will focus on these low-level system components in its analysis and its solution space. The remainder of this paper is devoted to finding solutions to maintaining the trust of applications dispatched from the nervous system to work nodes on the system. What is needed is an examination of hardware and low-level software in light of an untrustworthy operating system. Section 2 defines application clarity and a compositional operation for system clarity, as a formalization of what is meant by observable trust. We consider currently existing system architectures and discover the “trust and clarity gap”; a natural product of network trust models attempting to extend local trust onto a non-local system. The gap between high and low-levels on the remote system further suggests the importance of low-level system solutions. From these observations, section 3 proposes a taxonomy of possible solutions. Sections 4, 5, and 6 explore this solution space. Section 7 summarizes our work, considers hybrid solution models, and discusses the direction of further research.

2.0 The Trust and Clarity Gap

2.1 A Definition of Clarity

A process can be described by many properties, such as real-time guarantees, which relate the operation of a process to a system and the process itself. In this report we seek to maintain the clarity of applications running on untrustworthy systems. We formally define clarity below.

- **Application Clarity** over property P is the extent to which reliable observation that an application obeys or does not obey a liveness or safety property P can be obtained.

Informally, as an integral part of a system, a component with clarity of P is one that can, at time t, be reliably observed to determine its relation to P. This differs from the traditional notion of trust, in which trust is the degree of confidence in the consistent adherence (or violation) of a system to a property P. Informally, we define clarity of a system as the extent to which an observer can obtain the necessary information to compute trust of the system. Trust implies clarity, but clarity does not imply trust. That is, clarity is a necessary precondition for un-assumed trust.

2.2 A Sketch of Some Important Liveness Properties

We will be considering application clarity for several liveness or juxtaposed safety properties, defined as

- **Operational Liveness** indicates that an application:
 1. contains only code that the application author intended it to contain to run on a system;
 2. contains only the data that the application author intended it to contain to run on a system;
 3. maintains its contained data state such that the only actions having been performed upon it are those intended by the application code (including registers);
 4. is initialized in a manner semantically equivalent to the application initialization protocol agreed to (between the application author and the run-time system) prior to run-time instantiation of the application; and
 5. as a process, performs instruction streams, in serial or parallel execution, only as intended by (semantically equivalent to) the original application implementation.
- **Privacy Liveness** indicates that an application is maintained so as to receive correct enforcement of memory and data privacy services from a security policy on the run-time system, as stated to the application author just before the application is run-time instantiated. (includes register privacy)
- **Reservation Liveness** indicates that an application receives process scheduling and other resource allocations in a manner consistent with the policy of the run time system stated to the application author just prior to the run-time instantiation of the application. If any change is to take place in the scheduling and other resource allocation policies of the system to the application, they must be stated to the author prior to being put into effect.
- **Resource Assignment Liveness** indicates that an application receives the actual resources it requests, rather than spoofed resources, or alternate resources that were not intended by the application design.

We will be concerned with operational liveness, privacy liveness, fair reservation and assignment liveness clarity. We will utilize these forms of clarity to maintain an adequate level of trust for an application residing on the untrustworthy system-enough to meet our purposes in crossing the trust boundary in the first place.

2.3 Clarity Model

Let $\text{clarity}(P,A,t)$ be a measure from 0 to 1, where 1 is absolute knowledge that a liveness or safety property P is obeyed or not for application A up to time t . 0 is absolute knowledge that P is either obeyed or not obeyed by A up to time t . Intermediate measures are a confidence rating in application A obeying or not obeying P up to time t . Therefore we have two ways to utilize quantify clarity in a system. We can utilize a scalar measure from 0 to 1, or simply utilize Boolean clarity, where a system either maintains clarity (1), or does not (0.)

2.3.1 Clarity Composition

Given a system of components, we can compose knowledge of clarity for the system as a whole from the components. Let our model consist of n properties, for which we wish to model clarity propagation. Let each component i of our system depend upon the resources and services of components in the set S_i . Let the internal clarity of a compo-

nent (without considering dependences) be J_i . Let component i have clarity measure for property P up to time t of $I(P, i, t)$. Let it be that the clarity of property P_a is transferred from resource or service of component k to a dependant component by function $C_k(P_a, I_1, \dots, I_n)$ where I_1 to I_n are clarity values of the n properties at component k at time $t-1$. Then the clarity of property P_a at time t at component i is given by

$$I(P_a, i, t) = J_i \prod_{k \in S_i} C_k(P_a, i, I(P_1, k, t-1), \dots, I(P_n, k, t-1))$$

In words, a function C_k at component k describes how the clarity of each property at k just prior to our next clarity determination effects our clarity knowledge of our property for an observing application. Since this function is a variable of the observing component, the component being observed, and previous clarity values for all properties of our model, it is a function very specific to the individual system being explored.

We will also be interested in components which actively determine the clarity of other components. Such components will generally monitor other components to determine the status of some liveness or safety property P on the observed component.

2.4 Trust and Clarity Gaps

Consider the typical modern, layered system architecture, as depicted in figure 2. Each higher layer relies on lower levels and itself for services and resources. There is a direct correlation between intrinsic clarity, and the depth of a layer. Applications tend to have increased complexity over the more general services at lower levels. Also, higher levels tend to have more specialized services and resources, and therefore tend to not be as well tested and used. As a result, less clarity information is naturally collected about higher level functionality than lower level functionality. In summary, clarity tends to decrease with increasing dependence, complexity, and decreased testing or use. This leads to a natural trust vector on isolated computer systems- with each system layer looking for trust from the layers beneath it, which happily, correlates with the services and resources on which it relies and for which clarity information is more readily available.

In today's networked world, as depicted in figure 1, nodes cannot afford to assume the trust of one another. Figure 3 represents two nodes of a system. The left hand node represents a trusted dispatching node which will send a trusted application to run on a remote node. The right hand node represents the remote node. From the dispatching node's perspective (left node of figure 3), the dispatched application code is trusted, but the services on which it relies cannot be trusted at the remote (right hand) node.

The software of the remote node is assumed to be untrustworthy, as the network is large and faults (including attacks) will occur. This is the required model for the left node to be able to provide non-trivial, non-local fault tolerance for the remote system. For example, if the left hand node is to detect insidious deliberate faults, it must start with the assumption that the right hand software cannot be trusted. As a result, the left node can-

not rely on software on the right node to say “Everything is fine” as this reporting is untrustworthy. **Non-local clarity** requires some trusted component on the non-local machine.

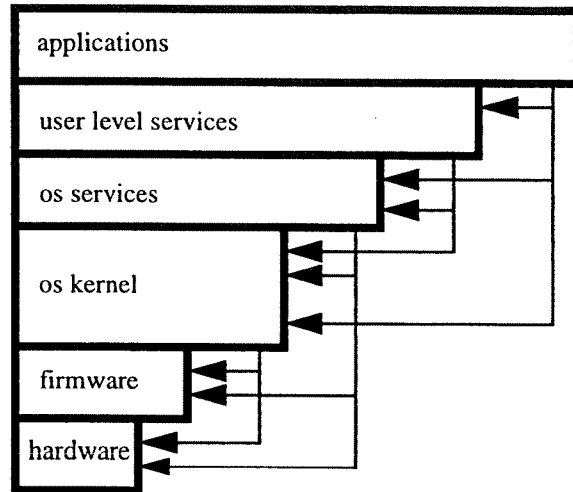


FIGURE 2.

The clarity and trust hierarchy of the typical system architecture. Arrows indicate service and resource dependencies between layers of the system.

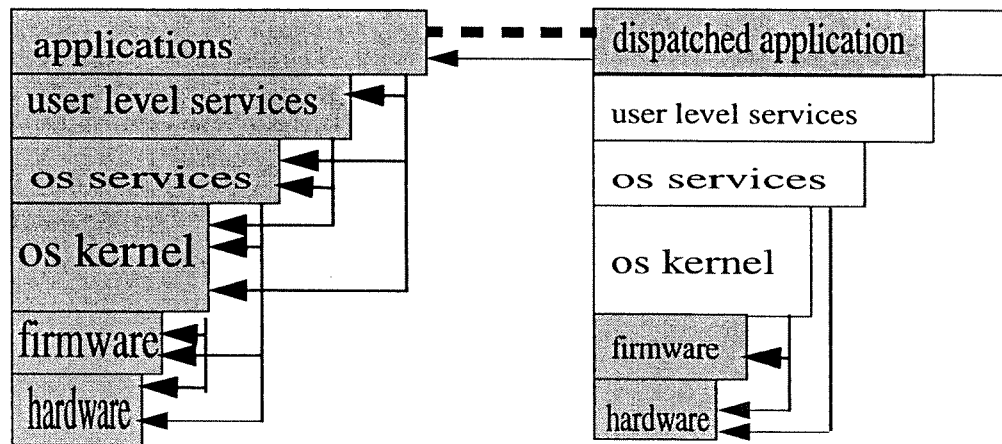


FIGURE 3.

Trust system model for a trusted dispatching (left) sending an application to an untrustworthy work node (right.) Shading indicates Boolean trust, with clear boxes indicating a lack of trust. Arrows indicate trust dependence. The dashed line indicates a trusted link between the two nodes.

The application is built and verified on the left hand node, and is thus a trustable component (internal clarity, $J=1$) before it is dispatched. The application, which must remain trusted, is now present on top of untrustworthy services and resources. In other words, a natural extension of the trust hierarchy on the local node (left) becomes unreliable when transported to utilize components at the remote node (right.)

This is unfortunate, because without trust for the dispatched application system, we must actively monitor the services and resources on which the application is dependant in order to maintain its trust. However, just as there is a trust gap, there is also an clarity gap for the application. It cannot monitor resources and services for clarity without relying upon these resources and services.

- We utilize the right hand node of figure 3 to visualize the **trust and clarity gap** scenario.
- It is clear that trust and clarity gaps occur naturally on an inherently untrustworthy distributed system with dispatched trusted applications.
- **Trust Relativity:** Figure 3 represents the standard large network scenario for trust, namely **trust relativity**. Each node trusts itself but cannot trust the resources or services of other nodes. We can strengthen our arguments by not assuming “global” or “absolute reference frame” of trust in the system. However, since our system maintains a trusted nervous system, we will often be able to eliminate trust relativity conditions.

3.0 Solution Taxonomy

In this section we will consider ways to maintain trust of the dispatched application in the presence of a trust and clarity gap. Our solution taxonomy will rely on the observation that an clarity gap (and hence trust gap) is a necessary precondition for untrustworthiness of a trusted application dispatched to run on a remote, untrustworthy node.

Assume that an application is prepared at the left hand node of figure 3. It is verified so that its internal components in composition are trusted. Thus we have full clarity of the internals of the program before we ship it off to the remote system. It is then dispatched to the remote node in the network over a trusted link. Since the behavior of the program when it receives its required resources and services is trusted, then on any remote node where services and resources are trusted, the execution of the application can always be trusted back at the dispatching node (from our trust composition operation.) It is precisely because we do not assume trust of a remote node’s services that we require clarity from the remote node’s services. In the presence of an clarity gap, we cannot establish trust. Hence a trust gap remains for levels of the system on which the formerly trusted application relies for trusted execution. As a result, the trusted application cannot execute in a trustworthy manner on the remote system.

Traditionally, there are two methods to keep a necessary precondition out of the picture: prevention, and avoidance. Detection is also an effective solution so long as reacting to a positive detection of a necessary precondition can lead to acceptable action to remove the precondition instance. We will divide up our solution space by solutions which we will define as trust gap prevention, trust gap avoidance, and trust gap detection.

- **Trust Gap Prevention:** This technique attempts to eliminate trust gaps for all remote nodes (and hence any clarity gaps.) This can be done by inserting our dispatched application at a base trusted level of a remote system, or by raising the level of trust of the remote system. Both techniques have been pursued by other research groups and are discussed in section 4. We investigate shortcomings of the Pentium and modifications to its system architecture that will help raise trust of address space protection to the application level.
- **Trust Gap Avoidance:** Trust gap avoidance relies on there not being an clarity gap on the remote system in order to dynamically compute trust of the software layers of the remote system. We will consider such approached in section 5. One solution is to have clarity local to the application when it arrives at the remote node, and allow it to determine trust of the system.
- **Trust Gap Detection:** Trust gap detection does not rely on the absence of an clarity gap. It asserts trust by running a remote trusted automata to detect trust safety violations. The monitoring device is attached to the highest level of the remote system with clarity (for our purposes, hardware.) It observes the behavior of the hardware to detect safety violations committed against our application from untrustworthy levels of software. By detecting any safety violations of untrustworthy layers at run time, trust of our remote application can be maintained. Trust gap detection uses lower level clarity without intermediate clarity. This is possible because necessary process resources for the monitor are maintained within the trusted monitor device.

3.1 Hardware Assumptions

In the following sections we will consider the advantages and disadvantages of each approach. Although only stated for one section, our implicit assumption is a system architecture similar to modern day commercial architectures such as the Pentium III system.

At some level we must trust a remote node. If we trust nothing of the remote node, then we have nothing at the remote site with clarity. We will make the assumption that the hardware, and perhaps firmware, at a remote node, are trusted. This assumption means that we will be primarily concerned with detecting trust gaps in the software components of the system, from the operating system kernel on up to the application level services. We will specifically state circumstances in which hardware is not trusted.

4.0 Trust Gap Prevention

Trust gap prevention relies on system properties that do not allow a trust gap to occur. We can prevent trust gaps by making certain that wherever we send a trusted application, trust can be maintained on all of the services and resources the application will utilize. Figure 4 shows the two general paths available in trust gap prevention. We can either raise the trust level up to our application, or lower our application to the existing trust level on a system.

There are natural advantages to trust gap prevention. It is efficient to be able to simply dispatch an application to a foreign system we know we can trust. In other words, we would have no need for clarity information at run-time. It is also simple, from the sys-

tem design perspective, in that application dispatching over secure links is sufficient to maintain application trust from the servicing node.

On the other hand, moving the application position in the hierarchy, or moving the trust boundary in a remote node may introduce difficulties in implementation, limitations in application, or security concerns for a node that receives an application from a foreign dispatcher.

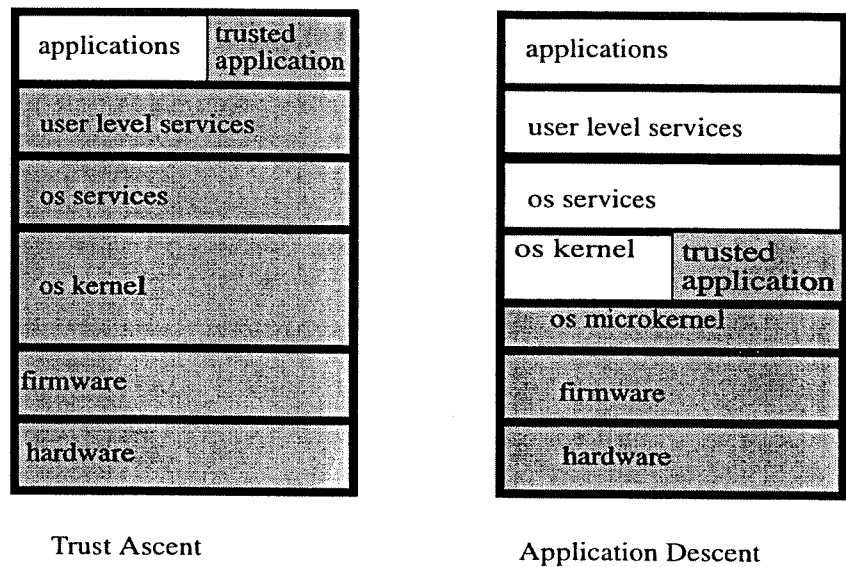


FIGURE 4.

An illustration of two methods for trust gap prevention. Trust ascent raises the inherent level of trust for all nodes to the boundary with the application level. Application descent lowers the level of a trusted application to interface the boundary of a trusted microkernel.

4.1 Clarity Ascent through Hardware Run-Time Trust Checks and Enhanced Clarity Capabilities

Static verification of core operating system trust properties has proven to be a difficult and expensive task. Establishing a trusted code base from static observation is the traditional method of trust ascent. However, it is expensive, requires full access to code in order to establish proofs, and requires reanalysis for non-incremental system changes. In other words, clarity of trust properties at system construction time is available, but expensive to utilize. In a heterogeneous reconfiguring network, COTS components and legacy code will form numerous combinations that may change rapidly over time. As a result, static proofs for a trusted code base are inadequate for our purposes.

In order to implement trust ascent to the application level, we must derive a technique for which clarity is less expensive to utilize, but which continues to guarantee that a trust

gap cannot arise as an inherent property of the system. Run-time checking and local fault tolerance for trust properties can eliminate the need to verify such properties at system construction time. Fault tolerance for run-time trust faults is beyond the scope of this work, but we can at least bolster clarity with run-time detection of trust faults. Software run-time checks are unlikely to be sufficient as they will sample at the rate of execution of the checking software. They are also subject to tampering. Our solution for clarity ascent is to augment hardware to guarantee trust properties at run-time which cannot be guaranteed statically or in software with sufficient speed or clarity. While we are modifying hardware, we can also provide mechanisms to enhance run-time clarity for an operating system.

We will illustrate this approach by guaranteeing properties of a hardware protection mechanism. Memory protection is an important portion of operational, privacy, resource reservation, and resource assignment liveness properties. Without memory protection, we cannot guarantee operational clarity of code or data state for a process. Likewise, privacy requires memory protection disallow accessibility between processes. Resource reservation and correct assignment require that assignment tables maintained in memory are adequately protected.

The Pentium processor, as with any current general purpose processors, relies on the operating system to set up and control data tables utilized by the hardware for protection mechanisms. Figure 5 shows critical components of a typical system required to establish protection trust for an application, with focus on the basic model of the Intel P6 processor series system. Arrows point from a component towards those on which it relies.

- Notice that almost all components rely on memory (and register) protection in order to be protection trusted.

At the bottom of the graph are the data tables that the hardware requires in order to function correctly. Memory protection requires these data tables to be trustable. However, for these data tables to be trustable, memory protection (from the operating system) must be trustable.

What we observe is that there is a circular trust dependence between the memory protection mechanisms in the operating system, and the hardware-required protection tables that the operating system must maintain.

In the typical isolated system scenario (no network) the memory protection system is trusted as well as the hardware-required data tables. As a result, component trust easily propagates to higher level components. We cannot afford to make such assumptions about non-local node operating systems. Therefore, we will augment the Pentium hardware with:

- an isolated kernel address space
- hardware run-time check rules

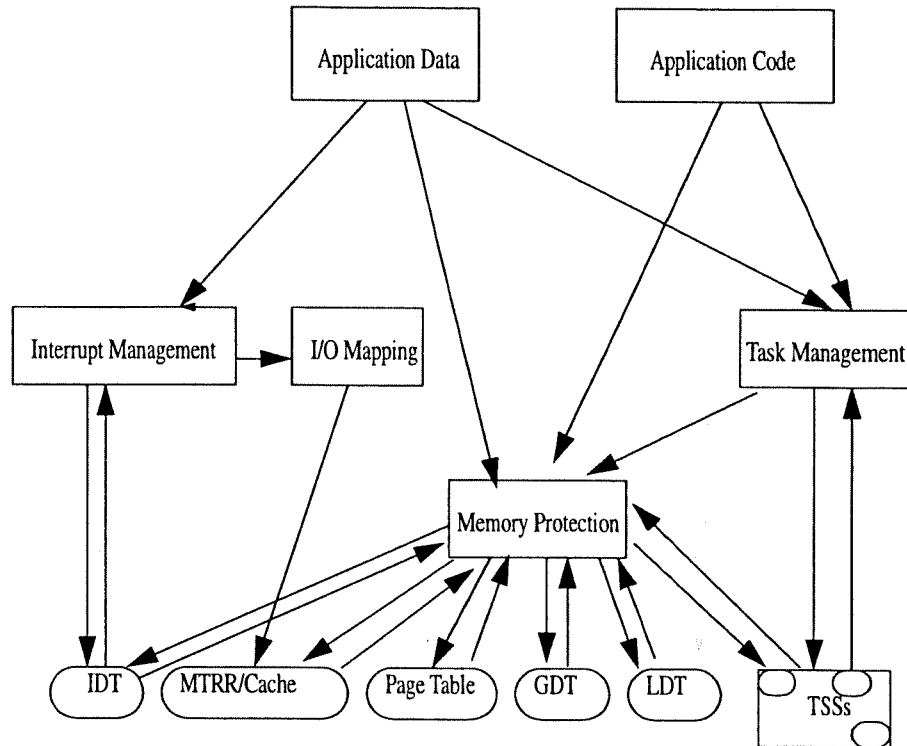


FIGURE 5.

A simplistic trust dependence graph for basic P6 system components. An arrow is drawn from the trusting object to the object that is to be trusted.

4.1.1 Isolated Kernel Address Space

The Pentium maintains a separate address space for I/O. We will extend this concept by maintaining a separate address space for the operating system kernel. The operating system kernel will be defined by any code that is running at privilege level 0 (most privileged.) The address space contains two components.

1. A ROM code address space
2. A RAM data address space

Access to address spaces is defined as follows:

- A process with current privilege level is 0 will fetch instructions from the separate ROM address space.
- A process with current privilege level is 0 will fetch data from the separate RAM data address space if the address range is within a hardware prescribed range (the upper 2 GB of a 32 bit physical address.)

- Any process with current privilege level greater than 0 will always access code and data from the regular address space.

Thus access to the kernel address spaces is an implicit operation. We have implicitly isolated a kernel code and data space. We have also intrinsically forced the code space of the kernel to be execute only, even to the kernel processes.

Kernel-Only RAM space is an address range of RAM that represents a different RAM bank for level 0 processes than other higher level processes. It can only be accessed by privilege level 0, kernel mode operations. The immediate effect of this limitation is that regardless of protection table information, only kernel mode process can modify this memory region. The hardware-required data tables are to be put in this address range, as well as other critical kernel data information.

It is assumed that the Kernel Only RAM space be large enough to accommodate all TSS and page tables the system may require, as well as space for other kernel data.

At the same time, this introduces a data address range in the standard address space that is inaccessible to directly read or write by the operating system kernel although, it is free to allocate or deallocate it.

4.1.2 Additional Memory Protection Data and Run-Time Checks

In addition to the address spaces presented above, the hardware can be extended to perform checks on what sorts of processes have accessed given memory regions as follows:

- Hardware data table access faults if the table is outside the kernel RAM address range.
- Page table entries and segment table entries maintain a bit that is set whenever NON kernel ROM code (level 0 processes) accesses a page.
- If ROM Kernel code accesses data in a page or segment that has been accessed by NON ROM Kernel code (the bit is set) a fault occurs.

Note that the kernel can clear the bit in a segment or page table entry in order to access the segment or page. The Pentium currently has a reference bit in page and segment tables. We merely wish to have a separate bit for reference by non-kernel level processes.

The run-time checks allow the processor to keep track of whether other memory regions of memory space have been accessed by non-kernel level processes. Furthermore, they prevent the kernel from “unknowingly” accessing memory that is assumed to be sheltered from non-kernel process access. In other words, to access memory that may have been accessed by non-kernel processes, the privileged mode process must first clear the segment or page table bit (in kernel RAM space) which indicates whether a segment or page has been touched by non-kernel processes. This increases the availability, with careful kernel coding, of run-time protection clarity information, allowing additional trust properties to be deferred to run-time maintenance.

4.1.3 Properties of Isolated Kernel Address Spaces and Run-Time Checks

Adding the isolated kernel address space and run-time checks provides us with the following run-time guaranteed protection trust and clarity features which do not exist in a traditional Pentium system:

- Only kernel processes may access or modify hardware-required table information. (Trust)
- Kernel code cannot be modified or read. (Trust)
- The kernel may store additional data in an address space only kernel level processes can modify. (Trust)
- Only the physically installed kernel code in kernel ROM can be executed as level 0 privilege code, and hence execute privileged mode instructions. (Trust)
- Instructions fetched from outside the range of legal kernel ROM by level 0 processes result in a memory access fault. (Clarity)
- Kernel processes cannot access data that has been modified by processes other than those of the kernel without at least acknowledging this possibility by clearing the page and segment table info of non-kernel process accessed bits. (Clarity)

Our run-time enforced trust and clarity mechanisms have reduced the burden of verification in operating system software design. So long as hardware is trusted, our additional checking can be trusted.

We cannot guarantee with run-time checks of our magnitude that kernel level data will not be attackable by stack smashing, and such, due to poor kernel or kernel interface programming. However, we can guarantee that no such attack will change kernel code. As a result, the most prominent problems of a kernel attack, namely replacement of kernel code with some other code to execute with privilege, has been eliminated. What remains is the possibility that poor kernel design will lead to abuse of kernel routines through abuse of the parameters or non-determinism errors in such functions. Kernel data might be corrupted in this way. If the protection tables are corrupted, protection could still be compromised for the system. Verification of kernel properties in a design and implementation is still required, though more properties are hardware trustable, and others can be made simpler to prove by use of our enhanced clarity checks. If such proof of algorithms and implementations in the static kernel ROM can be established, then memory protection can be trusted. The result would be propagation of component trust for memory protection to higher level components. The resulting picture of trust is as described in figure 6, in which trusted required dependencies are made bold. We can see that the hardware-required data tables can now trust memory protection, as it is enforced by hardware. As a result, there is no longer a circular dependence of trust between memory protection and the tables which enforce it. Apart from weakening attack strategies, we have made it simpler to prove remaining properties of memory protection.

Our architecture adjustments impose certain restriction on the way a kernel should be designed and interact with the surrounding system. For one thing, kernel level system calls on a Pentium maintain a level 0 stack for use by a process that passes through a call gate to level 0 privilege mode (this avoids task switch overhead.) A good implementation for trust propagation would utilize kernel RAM address range for the level 0 stack

of the process. The kernel can copy level > 0 stack information to the level 0 stack as needed. Furthermore, some system functionality that might fall outside a kernel by modern standards, such as some task management, might be required to be considered, once again, part of the kernel. This is due to the fact that privileged mode instructions can only execute from the ROM kernel code address space. It should be noted that the more run-time hardware trust bolstering we do, the more restrictive the operating system kernel design becomes.

We have described how trust can be ascended through modifications of Pentium hardware. However, we only demonstrated this for memory protection. Other required features, such as interrupt management, I/O mapping, and task management, are also important parts of operational and resource reservation and assignment liveness.

1. Attempt to establish hardware systems for system-design-time proof of additional component trust.
2. Attempt to establish run-time hardware systems to help manifest trust of these components at run-time.
3. Rely on external observation to determine some resource reservation and assignment trust. Some trust can be obtained by observing an output signature of the application over a trusted link to the dispatcher node. As an example, some trust of fair scheduling can be obtained by observation of a trusted link signal from the application.

One solution to the I/O mapping problem might be to have I/O mapping be committed in firmware to be unadjusted by software tampering. Current system BIOS's, which cannot be overridden by an operating system and are password protected would be adequate for this task. A more permanent I/O mapping would also be a solution. In the section on Trust Gap Detection we will discuss an important alternative to the techniques discussed here.

4.1.4 Conclusions about Trust Ascension

Trust ascension has the advantage that it does not require us to create special privileges for trust for our dispatched application on the remote node. It also does not upset the traditional balance of centrally based trust, relative to each node of the distributed system.

Our solution approach is to provide guarantees of trust properties at run-time through hardware, thus reducing the burden of proof from static code analysis and model checking of large and complicated operating system components. While we have demonstrated memory protection bolstering through run-time checks, higher level resources might require additional hardware to achieve the same effect, if trust propagation cannot be reasonably maintained through static proof.

Further investigation of this avenue, through analysis of operating system components by the trusted code base community, would be valuable. While they attempted to statically verify properties of trust, our approach, in summary, is to guarantee them at run-time through hardware and reduced software verification requirements.

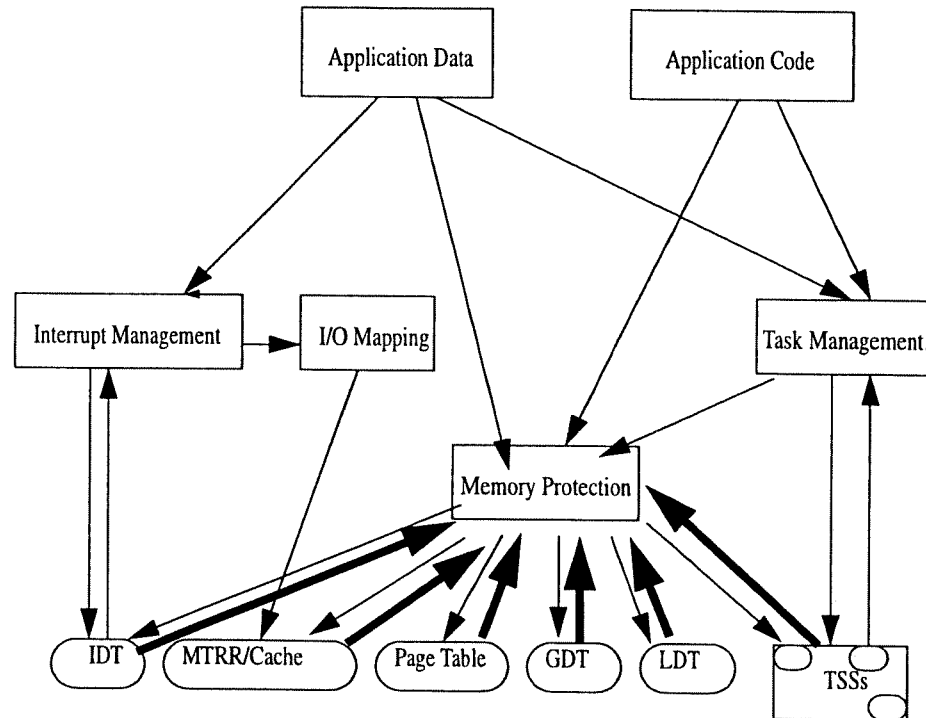


FIGURE 6.

A reworking of figure 4 for our modified architecture. Trust dependences which now carry trust are drawn as thicker arrows. We have eliminated the trust dependence cycle present in figure 4.

4.2 Application Descent through PCC Applications on a Trusted Micro-Kernel

In this solution, our goal is to have the dispatched application rest on the trust border at the remote system. The remote system must have a remotely trusted computing base consisting of a micro-kernel. Such a micro-kernel may be difficult to prove, but should only provide enough functionality to hand control of the hardware over to the trusted application, with a little abstraction of hardware resources. The micro-kernel would be maintained in ROM to avoid susceptibility to overwrite attacks. It is possible that we could simplify the task of trusting a micro-kernel through the hardware mechanisms of section 4.1.

The result of this application descent is that the application must be compiled to run at a level of abstraction and responsibility far beneath what a typical application must handle. This can be done at the trusted dispatching node, knowing the micro-kernel protocol at the remote node. The net result, however, is a net loss in application portability, and

the requirement of smarter, more elaborate compilers and linkers on the trusted node. We would also lose the ability to dispatch trusted COTS applications.

Furthermore, the micro-kernel would need to provide adequate resources and services such that the application could maintain operational, reservation and assignment, and privacy liveness properties. Such a micro-kernel would be required to allow the trusted application to schedule itself, or provide a scheduler within the micro-kernel. Likewise, either priority would need to be given to the application for interrupt assignment, or the micro-kernel would need to handle interrupt reservation assignment. In other words, the micro-kernel model would need to handle very little and “hand over the system” to the application. Otherwise, the micro-kernel would rapidly balloon into a complex traditional operating system.

Since the computer will be handing over substantial power to an application, at issue with this technique is the extent to which the incoming application can be trusted by the remote node. The remote node would require a sophisticated proof carrying code analysis technique in order to justify handing over low-level system capabilities to the incoming application. PCC is an active area of research/ However, if we allow that the network assumes trusted of the support nodes, then such proof carrying code is not required. Existing micro-kernel infrastructures, if trust verified, might be sufficient to implement this technique.

5.0 Trust Gap Avoidance

Trust gap avoidance relies on there being no clarity gap, in order to maintain trust for a dispatched applications required services and resources. Trust is maintained by actively monitoring the system layers for trust, using the available system clarity. Rather than seeking to detect trust faults, however, we seek to determine a proof of trust upon insertion of the application. In other words, upon receipt of a trusted application, a work node utilizes extensive clarity to guarantee to the dispatching node that trust will be maintained for the application.

One technique would be to have any services and resources required by the trusted application, such as fair scheduling, memory protection, registers, i/o, etc., be a required to present proof to the dispatching node, that they can be trusted. Such proof carrying resources and services, represent a solution parallel to the inverse problem of proof carrying code for untrustworthy applications. We do not discuss this technique further in this report.

6.0 Trust Gap Detection and Application Specific Trust Detection

Trust gap detection is an attempt to detect a trust gap when clarity at intermediate levels of a system hierarchy cannot be obtained or utilized efficiently. Instead we obtain clarity for higher levels of the system by monitoring system behavior at a low, trusted, level. If clarity for higher levels is not too expensive to extract from low-level state information, then we have a method of detecting trust violations (existence of a trust gap) at higher

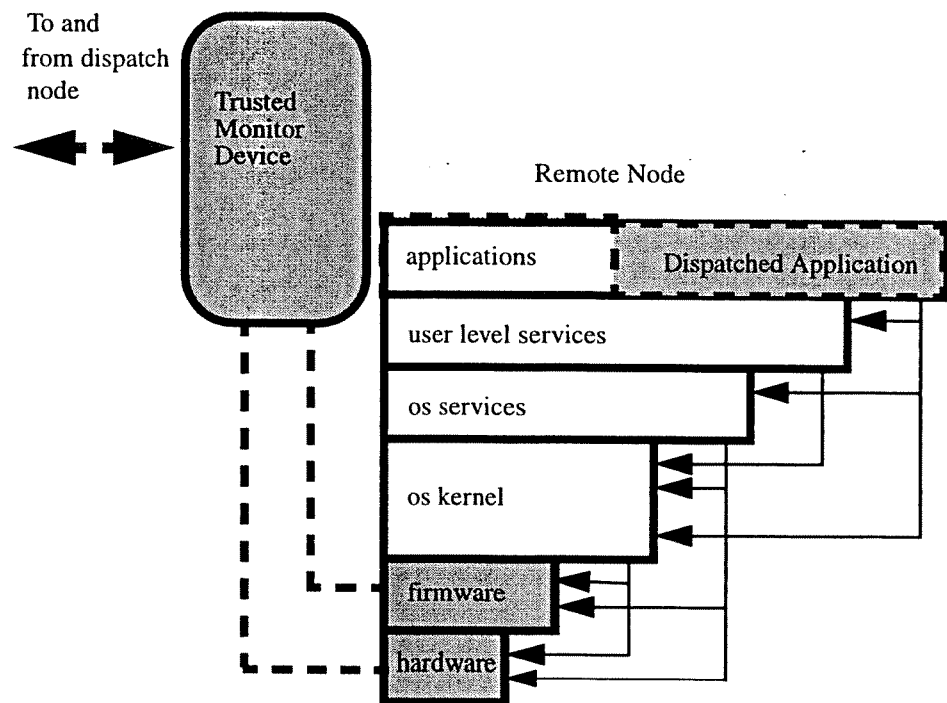


FIGURE 7.

A trust gap monitoring device is attached to a remote node, where a dispatched application is present. Trusted components appear with shading, while components for which trust-violations must be monitored are clear. Trusted links are drawn with dashed lines. Arrows indicate resource and service dependencies.

levels of the system. In this section we will demonstrate how observation of hardware state by a trusted monitor is sufficient to detect a wide range of higher-level trust faults.

Our approach requires that we extend monitoring of our dispatched application to beyond the application's output signature. We observe the behavior of the remote system more directly. Figure 7 shows the scenario that takes shape. A trusted monitor system is paired with the remote node. The monitor system might be a any form of trustworthy monitoring device. For a start, imagine a specialized embedded device with wires tapped into key components of the remote node system.

While figure 7 depicts the monitor at the hardware level, the monitor could sit at any level that exists before the trust gap where total system state can be observed within the level. Clearly, hardware is a prime example of such a system level.

Through the monitoring of the required trust properties of the dispatched application, we can determine whether, for example, memory protection trust is being maintained. Furthermore, we can determine whether I/O trust is maintained, as well as other components.

6.1 Available Data Streams

We can obtain the following data from the system quite easily

1. Contents of memory bus
2. physical I/O ports
3. external cache state and hits

It would be more difficult to tap into the following

1. Processor state
2. internal cache state and hits
3. TLB results and branch predictor results

Even further, we may desire to not only monitor state, but utilize hardware to actively search state. For example, we might want to occasionally take control of the memory bus so as to obtain a snapshot of some data region. (This might allow us to sample the memory bus less frequently than at every instruction.)

6.2 Protection Trust Fault Detection through Memory Reference Monitoring: Bus and Cache Snooping

If our trusted monitor is aware of the memory space devoted to our dispatched application, and we had access to the data bus as well as data and instruction caches on the Pentium system, then we could determine whether any other process violated the memory protection requirements of our dispatched process as follows:

- Watch all memory bus traffic and cache request hits and branch table hits.
- Parse the bus traffic as follows:
 - Determine a division of elements of the stream into instruction and data references.
- Use the following rules to detect memory protection trust inversion:
 - If a data access is to a region outside the application data area and the previous instruction reference is to an instruction within the applications instruction area, generate an untrustworthy data reference fault.
 - If a data access is to a region within the application data area and the previous instruction reference is to an instruction outside the applications instruction area, generate an untrustworthy data reference fault.

There are many complications to this model we have yet to consider. With the presence of growing and shrinking stacks, and dynamic heap usage, this becomes more difficult to clearly define, and we will need additional analysis of these circumstances. Also, virtual memory and run-time segmentation are problematic, as the monitor must know with certainty where the data and instruction memory of our dispatched process reside. If we could have our monitor determine the initial segment and page state of the applications memory, and verify page faults and such, then virtual memory is not a problem. However this may be beyond the scope of a reasonable experimental implementation. Segmented real-address memory is not as difficult a problem to tackle. Linked libraries, such as DLLs are a difficulty, as they append shared address spaces to an applications

instruction stream, and sometimes result in data space sharing. We also require that the monitor is aware of system calls, and of data sharing over system calls. The Pentium processor maintains separate stacks for separate protection levels, and thus data copying to system calls is an issue the monitor would need to be aware of.

Using this technique, we should be aware of any access to our trusted applications data, and which instruction stream is doing it! We even would know whether the instruction stream of our application is executing and getting a fair shake.

As with trust inversion prevention, we showed how memory protection can be rendered trusted. We also showed how clarity for applications can be increased. Just as with prevention, we leave as future work ways in which to show that other required resources, such as task and interrupt management can be trusted. If we have access to memory, it should not be hard to check the hardware-required tables of the remote node, to maintain trust for I/O and interrupts.

6.3 Processor State Monitoring

It may also be possible to monitor a system through the processor. We should be able to observe memory requests and register content to understand which process is executing instructions, and where data is being fetched from. However, since we would effectively only gain knowledge of registers over our previous model of bus snooping and cache snooping.

- Is there a reason to desire knowledge of general purpose registers?
- Would knowledge of segment registers, and the processor state flag register, yield additional useful information?

6.4 I/O Device Monitoring

There is no reason why we cannot extend our model to observe references to other devices on the computer system. For example, a hard disk drive plays a significant role in the storage of application executables and data files. Just as we monitor the memory bus for operations on memory that violate trust for an application, so can we monitor the commands sent to a hard disk drive, or other storage device, to detect violation of trust on the image of a trusted executable stored at an untrustworthy node.

Furthermore, with detection capability for temporary/permanent storage devices, we may be able to re-incorporate virtual memory into a system model supported by our monitoring device.

At issue is whether our monitor device can perceive enough information about the file system to maintain clarity of trusted application images and data on disk. Clearly, placing trusted applications and files within a fixed track or sector range, and declaring such a range as not to be written to, allows us to enforce write-protection trust for trusted process executable images. Clearly, we would need to have prior knowledge of operating system functions (and their code positions in a fixed kernel) that would be allowed to write trusted executable images into this disk space. Such considerations make a hybrid model of trust preemption with a fixed ROM kernel, and trust detection with a monitoring device, more attractive.

With sufficient knowledge of code that is allowed to read and write certain processes, we can guarantee read and write trust of trusted applications on disk storage.

6.5 Further Property Clarity with Active Memory Queries

We have demonstrated how passive monitoring can enforce protection trust for applications in both memory and more permanent storage systems. However, this is just the tip of the iceberg! Another possible monitoring technique is “active” memory probing, rather than just passive snooping of bus and cache activity. By placing the processor of a system and any other bus devices (such as DMA channelers) into a memory wait state, we should be able to take control of the memory bus, for short periods of time, without greatly affecting processor usage. As we will show, these techniques can help deter intelligent tampering and spoofing. In addition, they further bolster protection and I/O routing by allowing us to probe hardware required data-tables, such as an interrupt vector table or I/O mapping table.

6.5.1 Application Specific Property Clarity with Active Memory Queries

We have the opportunity to occasionally scan critical data and code regions of our trusted application in order to make sure they are intact. The following methods for active queries might be combined with the passive techniques given above:

- Provide the monitor device with an image of the application or of critical application regions. Also send the monitor information on how and what parts of these images to verify, and how often.
- Probe specific data regions and check data values to see that they are in accordance with the processes output.

For example, we might occasionally scan a critical loop of our application, and then occasionally check a loop invariant exit data value, or a loop variant data value that has some functional relation to program output. **Active memory monitoring allows us to monitor an application for compliance with static and dynamic properties (application specific properties) that are initially known only to the dispatching node and the monitor.** Complex properties of application provide the opportunity for the dispatcher and monitor to maintain a secret (which properties of the application are being monitored) that would render tampering less effective.

6.6 The Resulting Application Specific Detection Capabilities Are Strong

We will now suggest that a combination of passive and active memory/I/O bus and cache monitoring will be highly effective deterring tampering and faults with respect to our dispatched application. In fact, it will be highly effective in stopping spoofing and intelligent tampering attacks!

Our trust gap detection technique can be utilized to fight intelligent tampering and spoofing. We can have knowledge of relationships between our dispatched processes internal data state and its output, that only careful program analysis at the remote node could identify. Furthermore, in order to “spoof” output of the trusted application, the spoofer would need to also update the internal data state of the application in accordance with the relationship that we have knowledge of. Considering we have passive

monitoring to detect protection violations, this will be difficult for an adversary to conduct from the outset. In addition, we may utilize the code transform techniques developed by Wang, Hill, Knight and Davidson on our dispatched application to make static analysis (to determine invariant and variant relationships between internal process data state and output) more difficult or effectively useless. An intelligent tampering attack would have to be careful to maintain the relationships between the dispatched application's internal process data and process output. The adversary would be required to:

1. Determine which internal data the monitor is observing and correlating with which output of the application.
2. Get around the monitor's observation of data and code protection for the application.
3. Statically or dynamically analyze the application to determine the internal behavior of the application, and the relationships between internal data and output (to spoof the dispatcher node.) Also, information about important static relationships within the program must be understood in order to spoof a monitor observing application specific properties.
4. Create an alternate form of the application to spoof our dispatching node and monitor, while at the same time, updating the data state (and instruction stream reference) of the actual application. Or, manipulate the application so as to modify its behavior while not disturbing the properties of the application observed by the dispatcher node and monitor.

Since the information from step 1 will be private to the trusted application monitor and monitoring device, this will be difficult for an adversary to obtain. Step 2 will require the adversary to sneak past the monitor's passive observation of instruction and data streams. Step 3 will require the adversary to statically and dynamically analyze the program in an attempt to regain the information that the original author's have about the application. Since the monitor and dispatcher can change which properties of the application they utilize about the application, this will be extremely difficult. If we repeatedly change the internal details of the application and redispach it, this will be even more difficult for the adversary. Finally, step 4 requires that the adversary perform careful spoofing or tampering of the program.

6.7 Sampling and Processing Speed

One issue that must be considered is granularity of sampling over time. It must be assumed that the remote system operates at a particular cycle rate. Our monitor must be fast enough to not miss any cycle-to-cycle state information (in case of very short trust faults.) Our monitor must be at least as fast as, if not several orders of magnitude faster than, the instruction cycle rate of a processor in order to process passive information for the hardware state of the system. Meanwhile, our active monitoring can only occur occasionally, as it must take over the memory bus in order to probe memory. Careful timing with memory bus activity should allow more extensive memory probing with less disturbance of processes' use of memory.

6.8 Eliminating the Some Hardware Trust Assumptions

A robust trusted monitoring device with sufficient trusted links to a systems hardware might be able to detect a wide range of hardware faults. Although beyond the scope of this report, such detection capabilities are an advantage for fault-tolerant critical infrastructure concerned with more than just software faults. It should be noted that as we increase the number of trusted links to remote hardware the more robust the monitor device must be made in order to maintain trusted link to all monitored hardware.

6.9 Conclusions about Low-Level Trust Detection

We have suggested a model for low-level monitoring of hardware state and an algorithm for detection of memory protection violations from this state. This information allows us to detect any trust violation by levels of a system for which we have no clarity. We leave research into how other hardware components can be utilized to establish trust properties for future research. Furthermore, we demonstrated how active monitoring of memory can further aid detection of trust faults through monitoring of hardware-required data tables in memory. Active monitoring of memory also provides us with the opportunity to scan applications for violations of complex, application specific properties known, initially, to only the dispatcher and the monitor device. If the monitor's activity is not observable by an adversary, then an adversary cannot obtain the secret of which static relationships the monitor is observing. Since the number of application specific properties between code and data are large, and application implementations can be occasionally modified and re-dispatched, we have a technique for monitors to detect attempts to intelligently tamper with, or spoof, application behavior.

7.0 Summary and Conclusions

7.1 A Summary of Findings

We defined trust as the confidence we have about whether a property is obeyed or not. We defined clarity as the confidence we have about our ability to obtain information about trust of a property. We then showed how traditional system architectures lead to a condition we defined as a trust and clarity gap in a distributed application system, in which applications are dispatched from a trusted source node to an untrustworthy destination node. In our analysis we discovered two techniques to help eliminate the trust gap problem in dispatched applications over a large untrustworthy network.

The first is a form of trust gap prevention. It involved the modification of architecture rules to eliminate some of the cyclical trust dependences in today's hardware designs (in order to simplify the propagation of trust to higher levels on remote machines) where at least the hardware is trusted. The result is increased reliability of key trust requirements for foreign nodes that may be tampered with or are faulty at the software level. We demonstrated how this technique could be applied to the critical trust component of memory protection. We still require that kernel functionality be verified before being physically installed in the kernel execute only code space; but we have reduced the verification burden of the kernel. Future investigation can consider other crucial properties of application trust with relation to specific system trust goals.

The second is a form of trust gap detection. It involves the contractual physical placement of a trusted monitoring device at each remote node to which applications will be dispatched. It allows us to carefully detect memory protection violations, intelligent tampering and spoofing attempts, and potentially such behaviors as stack and buffer over-run attacks. In exchange for this monitoring, the dispatching node gains clarity information from a low-level of the foreign node, where sufficient state is present to detect a large number of trust faults. We can devise small monitors to detect only memory usage trust faults, or more extensive systems to detect trust faults on other devices, such as hard disks, as well. With active memory monitoring, we can even monitor faults in trust of application specific properties. This will help us provide trust against intelligent tampering and spoofing attacks. In general, monitoring for trust faults at a low-level allows us to detect a faults in adherence to a wide range of properties; from general trust properties to application specific properties.

Our detection technique does not depend on understanding “typical” or “atypical” data reference or instruction reference streams. Our monitoring criteria is formally specified by the internal implementation of the trusted application, or in the case of memory protection violations, simple rules of memory accesses by processes. It should be noted that a successful monitor device could eliminate the need for the dispatched application to maintain constant communication with the dispatching node to maintain trust. Thus, while sensors would be effective, “output quiet” actuators would also be instrumentable as trustable on an otherwise untrustworthy software system.

Our prevention and detection techniques defend against system errors, intelligent attacks, and critical software failures. It should be noted that trust gap prevention does not protect against hardware failure. Trust gap detection can defend against some hardware failures, when some still clarity-worthy piece of hardware contradicts the information of a clarity-less faulty hardware device. However, it cannot defend against intelligent tampering or spoofing at the hardware level. Thus our system provides extensive trust for many forms of software failure, and a good selection of non-adversarial hardware failures.

7.2 A Hybrid Approach

In combination, our two techniques may provide redundancy in providing application trust. In particular, the change in the nature of protection in hardware that we have suggested, through an isolated kernel space, is more of a revision on the model of trust in a pervasively networked world, than a specific technique for protecting remotely dispatched applications. It provides a new level of robustness at the hardware level, for general trust providing capability of networked machines with dispatched applications.

On the other hand, our monitoring technique provides very specific solutions to the question of trusting applications dispatched to other nodes where software is not trusted. Together, our trust gap prevention can help bolster the ability of the trust gap detection monitor to easily detect trust faults. Also, we have many remaining trust properties that we did not provide proof of solution by to either of these techniques. Depending on the power of trust gap detection, more or less trust ascension techniques may be required in order to make trusted applications relying on untrustworthy system software a reality.

As we saw when examining hard disk trust fault detection, having more knowledge of a fixed operating system makes determination of some faults less expensive.

7.3 Conclusions

We have identified two techniques to significantly reduce the problem of trust and clarity gaps for trusted applications distributed to systems with untrustworthy software. The result is the ability to create more responsive, fault tolerant network infrastructures, capable of reacting to non-local faults in a rapid, intelligent manner. In the short term, we have provided a method for bolstering trust of applications in any environment in which hardware is trusted. In an increasingly networked computing environment, issues of software trust will continue to arise, and the nature of who is trusted and who is not, will more readily make apparent the need for trusted software to run on systems where other critical software is inherently untrustworthy.

8.0 References

- [1] Amoroso E. Nguyen T. Weiss J. Watson J. Lapiska P. Starr T. "Toward an Approach to Measuring Software Trust," Proceedings. 1991 IEEE Computer Society Symposium on Research in Security and Privacy, IEEE Computer Society Press, Los Alamitos, CA, USA, 1991, pp 198-218.
- [2] Di Vito, BL. Palmquist PH. Anderson ER. Johnston ML. "Specification and Verification of the ASOS Kernel," Proceedings. 1990 IEEE Computer Society Symposium on Research in Security and Privacy, IEEE Computer Society Press, Los Alamitos, CA, USA, 1990, pp. 61-74.
- [3] Forrest, S. Hofmeyr, SA. Somayaji A. "Computer Immunology" Communications of The ACM, Vol 40, No. 10. October 1997. ACM, 1997, pp 88-96.
- [4] Genier G-L. Holt RC. Furkenhauser M. "Policy vs. Mechanism in the Secure TUNIS Operating System," Proceedings. 1989 IEEE Symposium on Security and Privacy. IEEE Computer Society Press, Washington, DC, USA 1989, pp. 84-93.
- [5] Hu AJ, Vardi MY. "A Formal Method Experience at Secure Computing Corporation," Computer Aided Verification 10th International Conference, CAV'98, Springer-Verlag, Berlin, Germany, 1998, pp. 49-56.
- [6] Johnston M., Stiriou V. "Testing a Secure Operating System," 13th National Computer Security Conference. Proceedings, Vol. 1, pp. 253-65.
- [7] Keedy, JL. Vosseberg, K. "Persistent Protected Modules and Persistent Processes as the Basis for a More Secure Operating System," Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, IEEE Computer Society Press, Los Alamitos, CA, USA, Jan 1992, Vol. 1, pp. 747-56.

- [8] Knight JC, Elder MC, Du, X. "Error Recover in Critical Infrastructure Systems," Computer Security, Dependability, and Assurance: From Needs to Solutions. IEEE Computer Society, 1999, pp 49-71.
- [9] Knight, JC, Sullivan KJ, Elder MC, Wang C. "Survivability Architectures: Issues and Approaches," Proceedings. Darpa Information Survivability Conference and Expose. IEEE Computer Society, 1999, Vol 2, pp 157-71.
- [10] Korelsky T. Sutherland D. "Formal Specification of Multilevel Secure Operating System," Proceedings of the 1984 Symposium on Security and Privacy, IEEE Computer Society Press, Silver Spring, MD, USA, pp. 209-18.
- [11] Kozyrakis, CE, Patterson DA. "A New Direction for Computer Architecture Research," Computer, November 1998, IEEE Computing Society Press, Los Alamitos, CA, USA, 1998. pp 24-32.
- [12] Sullivan KJ, Knight JC, Du, X, Geist S. "Information Survivability Control Systems," Proceedings. 21st International Conference of Software Engineering. IEEE Computer Society, Los Alamitos, CA, USA, 1999, pp 184-192.
- [13] Waldhart NA. "The Army Secure Operating System," Proceedings. 1990 IEEE Computer Society Symposium on Research in Security and Privacy. IEEE Computer Society Symposium on Research in Security and Privacy, IEEE Computer Society Press, Los Alamitos, CA, USA 1990, pp. 50-60.
- [14] Wismuller, R. Trinitis, J. Ludwig, T. "OCM - A Monitoring System for Interoperable Tools," SPDT 98, Welches OR, USA. ACM, 1998. pp 1-9.

APPENDIX D

“A System for Experimental Research in Distributed Survivability Architectures”

A System for Experimental Research in Distributed Survivability Architectures¹

John Knight, Robert Schutt, Kevin Sullivan

University of Virginia Department of Computer Science

Thornton Hall, Charlottesville VA 22903

Phone: (804) 982-2216, Fax: (804) 982-2214

E-mail: (knight | rschutt | sullivan)@Virginia.EDU

Abstract: The survivability of critical infrastructures is not assured today because of their reliance on complex, vulnerable information systems. Survivability enhancement will require changes to system architectures. Experimental systems research in this area is complicated by the private ownership, criticality, and complexity of infrastructure systems. A key research method is therefore to explore and evaluate novel architectural concepts by prototyping them within dynamic models of infrastructure systems. We present a toolkit to support the construction of such models and the prototype-based evaluation of novel architectural idioms. The toolkit captures common features of infrastructure systems that challenge the state of the art in survivability management. Our models and prototype constructs run on workstations running Windows 2000 and function as distributed systems of programmable, message-passing processes with reflective capabilities. We have built infrastructure models with up to 20,000 processing nodes, and we have evaluated several novel architectures, including one based on distributed hierarchical control.

Keywords: Survivability, testbeds, distributed systems, development environments.

1 Introduction

The survivability of critical, information-intensive infrastructure systems, such as electric power generation and control, banking and financial systems, telecommunications, and air traffic control systems, has emerged as a major national concern [16, 22]. By survivability, we mean the ability of such systems to continue to provide acceptable levels of service under predefined adverse circumstances [11]. *Acceptable* service is defined in terms of the aggregated impact of service disturbance to users who depend on the system. Such service might be a degraded form of normal service or an alternate service, differing in whole or in part from normal service, as determined by application experts and policy makers [20]. *Adverse* circumstances might be widespread environmental damage, major equipment failures, coordinated security attacks, and so on. In this case, survivability can be thought of as requiring fault tolerance with very specific and usually elaborate requirements for continued service. Novel forms of error detection, damage assessment, and so on are specific to the context of a complex distributed system.

Our concern is with survivability in the face of disruptions to the vulnerable *distributed information systems* that automate and operate the infrastructures. Tremendous efficiency improvements and service enhancements have been made in recent years in many infrastructure systems by the introduction of sophisticated information systems. In some cases these changes are very visible, e.g., the worldwide access to funds provided by ATM machines. In other cases they are not, e.g., just-in-time delivery of manufactur-

1. Revised August 2000.

ing materials by freight rail. The high-level problem is that enterprises, including civil and military organizations, are put at risk by vulnerabilities of critical infrastructure systems to disruption of their information systems in a world of increasingly complex information systems and increasingly capable adversaries.

Our research goal is to design survivable systems and to enhance the survivability of existing systems against threats that pose risks to the provision of acceptable levels of service. Achieving this goal in practice will require major investments in research, development, and re-engineering. We view survivability as a property that is made attainable, at least in part, by a system's architecture. Thus, we seek to explore, develop and evaluate innovative architectural techniques for achieving survivability in complex new and existing systems.

The survivability enhancement of existing systems is complicated by their size and complexity and thus their resistance to change. We therefore seek survivability enhancement techniques that can be imposed largely transparently on existing systems. The ability to do this is constrained, of course—often heavily—by the particular properties of any given system.

Experimentation is a crucial element of research in this area. It is vital, for example, to the early evaluation of novel survivability techniques. Unfortunately, several factors present serious impediments to experimentation. First, infrastructure systems are privately owned. Second, they are, by definition, critical from both business and societal perspectives. It is inconceivable that their operators would permit experimentation on them. Third, infrastructures are enormous in physical scale, cost and complexity, which makes it infeasible to replicate them in the laboratory.

We have thus adopted an approach based on the use of *operational models* of infrastructure systems as test-beds for developing and evaluating prototype architectural mechanisms for survivability. In this paper we describe a system that permits a wide range of representative models of critical infrastructure systems to be built rapidly and made the subject of experimentation. Provision is made within the system for creation, manipulation, and observation of models of infrastructure systems as well as the introduction of architectural elements designed to enhance survivability. The ability to develop and analyze models and prototype survivability mechanisms rapidly is an important aspect of the work because we wish to explore a range of infrastructure applications and a variety of architectural concepts efficiently.

The rest of this paper is organized as follows. Section 2 discusses the basic modeling concepts. Section 3 presents key aspects of infrastructure applications. Section 4 discusses the requirements for a modeling system. Section 5 presents our system in detail. Section 6 summarizes an example model that we have been built using the modeling system. We discuss related work in Section 7. Section 8 presents our concluding remarks and plans for future work.

2 Operational Models

Given the impediments to direct experimentation with real infrastructures, we have adopted an experimental approach based on operational models. By an *operational model*, we mean a simplified version of the real system that executes as the real system does—as a true concurrent system—but that provides only a restricted form of its functionality. The goal for a given model is to have a simplified laboratory version of the associated infrastructure system that is made manageable by implementing only relevant application functionality and implementing only essential characteristics of the underlying target architecture. Most infrastructure applications, for example, are distributed and this is an essential characteristic, although the particular protocols used in the underlying network are not in most cases. Thus for our purposes an operational model needs to be truly concurrent but it need not use any specific network protocol.

Building operational models of critical infrastructure information systems presents two significant challenges: (1) modeling the critical and no other aspects of infrastructure systems with sufficient accuracy and completeness; and (2) facilitating inclusion in the model of relevant architectural mechanisms to be developed or evaluated. For purposes of experimentation, an operational model has to represent relevant functional and architectural features of a given system, as well as its operational environment, including a dynamic model of vulnerabilities, internal failures and external threats. Once such a model is built, mechanisms must be present to allow prototypes of architectural survivability mechanisms to be introduced. Both models and architectural supplements must be instrumented for collection of data needed to analyze and evaluate survivability mechanisms.

The *prima facie* validity of conclusions derived from studies based on models and prototypes depends on the extent to which they capture the relevant properties of modeled situations. Validation of such results by other means is essential. Our results are not validated to the degree required to warrant adoption into real infrastructure systems. Rather, at this early stage of scientific inquiry into the critical area of infrastructure survivability, our work is at the level of basic experimental systems research, in both the technical and methodological dimensions.

3 Relevant Features of Critical Infrastructures

The design of the modeling system is based on two sets of issues. The first centers on key, relevant features of critical infrastructure domains determined through our in-depth study of several critical infrastructures systems [10]. The second centers on the need to enable experimental architectural research in largely transparent monitoring and control. In this section, we summarize the key domain properties; in the next section, support for experimental research.

The features of the infrastructure domain that we selected for explicit representation in our models were derived from the research questions with which we are concerned. An important aspect of this selection is that the open research problems are posed by precisely those properties that distinguish infrastructure systems from other, more familiar, systems for which high levels of dependability are required. The properties with which we are most concerned are as follows:

- *System Size*
The information systems supporting critical infrastructure applications are very large, both geographically and in terms of numbers and complexity of computing and network elements. For example, the banking system of the United States includes thousands of nodes in many business situations. Infrastructure systems are also widely distributed, in large part because they must deliver service streams over large, sometimes even worldwide, geographic regions.
- *Hierarchic Structure*
Many of these systems are structured hierarchically, although “short circuits” are sometimes present for performance reasons. The United States financial payment system, for example, can reasonably be viewed as roughly tree-structured with the Federal Reserve System at the root of the tree. In reality, the Federal Reserve competes with other financial institutions to provide many important services, such as check clearing, so a forest is perhaps a more accurate description of the payment system. When interdependencies among infrastructures are taken into account (e.g., the dependence of banking on electric power and *vice versa*), the structure of the infrastructure system-of-systems is seen as a complex nested hypergraph.
- *Serial Functionality*
At the highest level many of these systems operate as loosely coupled subsystems, each implementing a function that provides only *part* of the overall service. The complete service is only obtained if sev-

eral subsystems operate correctly in the correct sequence. For instance, the U.S. payment system operates through computations at local branch banks, centralized money-center banks, and very centralized clearing organizations, including the Federal Reserve Bank. In most cases, at least one bank at each level has to operate to clear a check.

- *COTS and Legacy Components*

The information systems underlying critical infrastructures are and will continue to be built from commercial off-the-shelf components, including standard hardware, operating systems, network protocols, database management systems, job control mechanisms, programming environments, and so on. These systems also include custom-built software, much of it of a “legacy” nature. That is, the software has grown and changed over many years, has a degraded structure, and is thus hard to understand and costly to change.

- *Multiple Administrative Domains*

Many infrastructure information systems span multiple administrative domains. For example, the U.S. payment system is an extremely complex system of systems for managing transfers of value and commitments to transfer value among financial and other institutions. The institutions include not only local and money-center banks as already noted, but credit card issuing and clearing organizations, check clearing organizations, loan organizations, financial and other exchanges, and so on. Moreover, in traditional physical infrastructures, such as transportation, computing systems are becoming integrated across corporate boundaries to achieve previously impossible supply chain efficiencies.

- *Availability*

The requirements for availability in infrastructure systems are considerable as would be expected; but availability is a more complex issue than it might appear. In some cases, availability requirements vary with function. It is important in power generation, for example, to maintain power supply if possible, but it is not necessary to maintain an optimal generation profile, and some customers can tolerate some interruptions. In other cases, availability requirements vary with the type of application node. In the banking system, for example, there are many branch banks but very few banks permitted to transfer value between organizations. Thus the availability of service at a local bank is relatively unimportant but it is essential for banks conducting value transfer. Finally, availability requirements vary among infrastructures, and they certainly vary among customers, and even over time. Longer-term power outages are more critical to hospitals than to homes, and in winter than in summer.

- *Complex Operational Environments*

The operating environments of critical infrastructures are of unprecedented complexity. They carry risks of natural, accidental, and malicious disruptions; sometimes highly variable loads; varying levels of criticality of service; and so forth. For example, freight rail service is especially critical in October—harvest time. Moreover, operational environments are now believed to have potential to exhibit previously unrealized behaviors such as widespread, coordinated information attacks. Cascading failures of infrastructure systems, in which the failure of one node leads to the failure of connected nodes, and so on, are also a real concern.

4 Modeling Requirements to Support Architectural Research

The general requirements for a modeling system derive from the two challenge areas identified earlier: modeling of infrastructure applications and modeling of architectural supplements. In addition, of course, support for experimentation has to be provided to allow for model control and display of results. The details of the requirements derive from the features of critical infrastructures discussed in the previous section, the architectural research goals, and the experiments that are to be performed. In this section, we outline the requirements that the system is designed to satisfy.

4.1 Infrastructure Application Modeling

The requirements for modeling the application infrastructure break down into four categories:

- *Application target architecture.*
Support must be provided to model distributed systems with large numbers of nodes. Heterogeneous nodes need to be supported along with arbitrary application network topologies. Since infrastructure applications typically make provision for availability, it must be possible to model all forms of redundancy. Finally, the model must allow rapid change of any element of the target architecture so as to permit a wide variety of systems to be modeled.
- *Application functionality.*
Crucial aspects of application functionality must be modeled. This includes typical processing that takes place on a single node and serial functionality where a series of nodes operate on a data stream to provide a single service to the user. In the infrastructure applications that we have analyzed, a great deal of functionality is associated with manipulation of databases and generation of associated reports. This does not mean, however, that the system needs to provide some form of elaborate database support. In fact, it is such details that have to be abstracted away to produce useful models of tractable size. Database issues associated with a single node are not relevant for our modeling purposes except in so far as they affect system-wide issues.
- *Application operating environment.*
All relevant input sources and output sinks have to be modeled. Input sources include all types of user requests as well as all forms of application data. Output sinks include displays, reports, application data, etc.
- *Application failures.*
All relevant types of failure have to be modeled along with all relevant failure parameters. Types of failure include hardware failure, software failure, operator mistakes, environmental trauma (e.g., hurricanes), security penetrations, and so on. Of particular importance are failures that affect large parts of the system such as coordinated security penetrations and local failures that cause cascading effects through the network. Failure parameters of interest include timing, scope, duration, extent, and so on.

4.2 Architectural Supplement Modeling

The architectural concepts of particular interest to us focus on monitoring and control [20]. This requires that sufficient state information be available to permit system-wide errors to be detected and system-wide damage assessment to be undertaken. In addition, we wish to implement continued service for applications in which many nodes cooperate to provide functionality, and in which continued service therefore requires the manipulation of the system as a whole. These general notions lead to the following three basic requirements:

- *Application information.*
It must be possible to acquire information about both running and corrupted application elements. This requires the ability for applications to supply prescribed information and the ability to acquire application information by observation. The acquisition of application information should be by means similar to those that might be used in a production implementation of the architectural ideas being explored. A key aspect of this requirement is that the facilities be transparent to the application to the extent possible. Thus architectural technologies related to the idea of “transparent wrappers” are of particular significance.
- *Application enhancement.*
It is necessary to be able to introduce enhanced functionality over and above the required application functionality. Enhanced functionality might have to be added at the network link, computing node, subsystem, or complete system levels. Again, this requirement has to be met by the modeling system in

the same manner as will occur in a complete application.

- *Reconfiguration.*
Reconfiguration of application architectures is a significant aspect of the way in which continued service is likely to be provided in infrastructure systems recovering from a failure. The modeling system obviously has to support this in as general and flexible a manner as possible.

4.3 Support for Experimentation

Since the purpose of the modeling system is to enable a variety of experiments to be performed, support for experimentation has to be provided, specifically:

- *Model control.*
Typical models that might be used for architectural research will involve large numbers of nodes and such models will be executed on diverse physical platforms. Provision must be made to control models in terms of mapping models to platforms, model initialization, induction of failures, and so on. These facilities are especially important for complex models that operate on target systems with large numbers of computers, some of which might be geographically remote.
- *Data acquisition and display.*
Data capture and the display of raw and processed data is essential. But it is difficult with large operational models of the type we require. The difficulty arises from the volume and the asynchronous nature of the data that might be generated in an experiment. A typical model will almost certainly involve thousands of application nodes, each of which might generate a data stream such as statistics on local network message traffic. For purposes of analysis, it might be necessary to process this data centrally and have ordering information so as to be able to predict recovery actions.

5 The Modeling System

5.1 Overview

The system we have developed to meet the various requirements outlined above is called RAPTOR. For purposes of experimentation, the RAPTOR system provides the user with an efficient, easily manipulated operational model of a distributed application with extensive control, monitoring, and display facilities. Figure 1 provides an overview of the system.

A RAPTOR model is specified by defining the desired *topology* and the desired *application functionality*. From the topology, the model is created using services from the modeling system's support libraries and using application software provided by the model builder. *Vulnerabilities* to which the model should be subject are defined and controlled by a user-defined vulnerability specification. During the execution of a model, *symptoms* can be injected into the model to indicate any event of interest to the user. Events might include security penetrations, hardware failures, etc. Any *data* of interest to the user can be collected and made available to a separate process (possibly on a remote computer) for *display* and analysis. Finally, since multiple independent models can be defined from separate topology specifications, complex systems of *interdependent* critical networks can be modeled (see Figure 2).

5.2 Building Blocks for Models

The basic semantics of a model is a set of concurrent message-passing entities that we refer to as *virtual message processors*. Figure 3 depicts the general structure of a virtual message processor. A virtual message processor is provided with a queue of incoming messages that it can read and process as it chooses. Usually, these messages are routed from the input queue to programmable message interpreters. Any new messages generated as a result of interpreting received messages are sent immediately although their

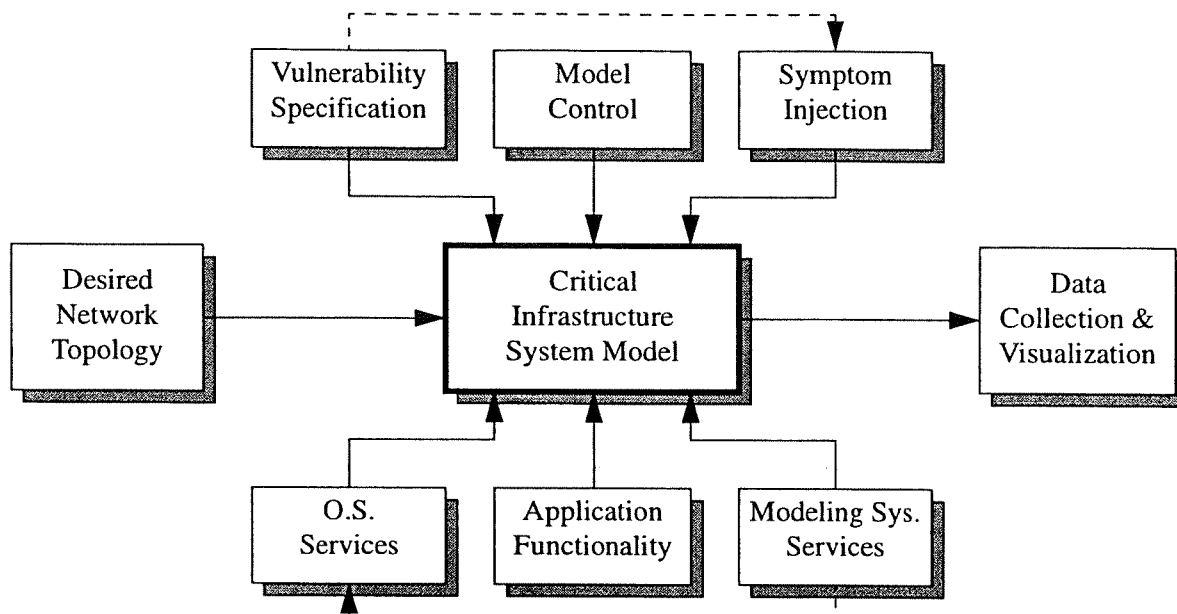


Figure 1. The RAPTOR modeling system architecture

arrival times at their destinations can be controlled. Messages can be generated asynchronously also if needed based on, for example, a timer event.

Within the modeling system, a network node is modeled as a set of one or more virtual message processors each of which is executed by a separate OS-level thread. A typical simple node can be modeled with a single virtual message processor and hence a single thread. More complex nodes can be modeled as a collection of threads thereby allowing such nodes to exhibit concurrent internal behavior. The use of threads for modeling nodes allows the model of a single node to be itself concurrent thereby permitting issues such as synchronization errors and race conditions to be modeled realistically.

Node-to-node communication is modeled by message passing between threads. Messages are passed through memory and so message passing is very efficient. Any thread can send a message to any other thread subject to restrictions imposed by a model's topology (see below). In order to permit models in which resource contention might occur, the input message queue for a virtual message processor can be

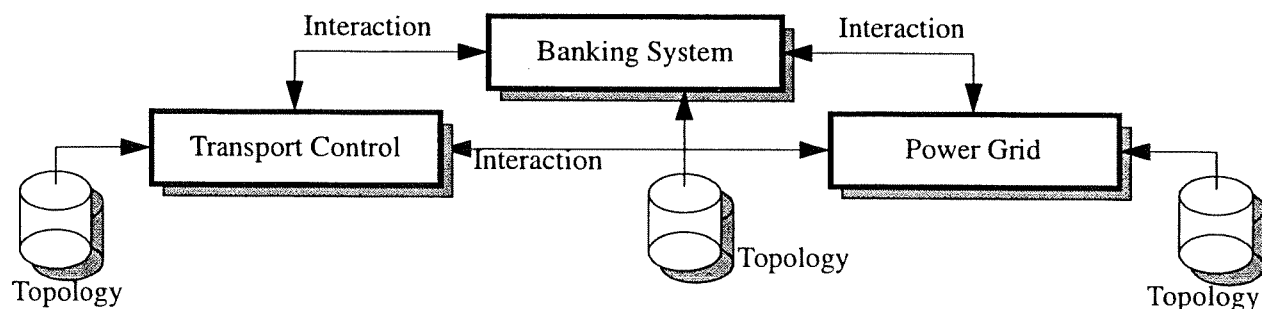


Figure 2. A Model of Multiple Interacting Infrastructure Systems

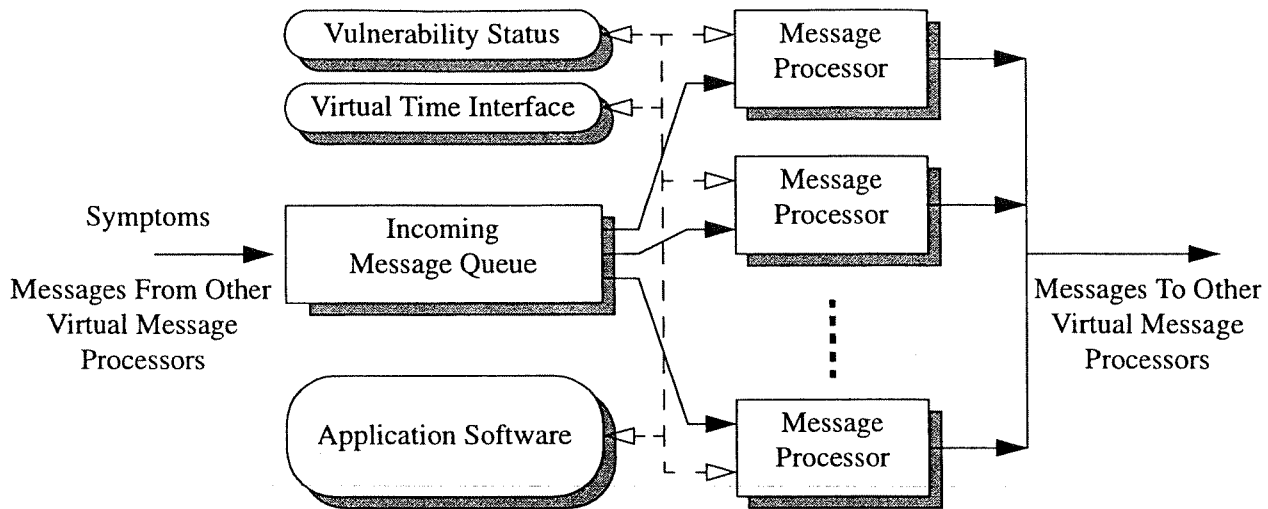


Figure 3. RAPTOR Virtual Message Processor

given a maximum size. If the input message queue is full when a message arrives, the attempt to send the message to that virtual message processor fails. Message transmission times can also be specified to allow transmission delays to be modeled (see section 5.5).

5.3 Model Topologies

The description of the model to be run is defined in a topology file. The topology defines precisely what nodes are to be created, what software each will run, and what the connectivity is to be. By using this approach to the definition of models, different models can be created quickly and modifications to models can be effected easily.

The topology specification defines the required virtual message processors that are to be run so it is a simple matter for the user to arrange for a node in the model to consist of any desired number of virtual message processors. The specification of connectivity is unidirectional, i.e., it states that one virtual message processor may send messages to another. By combining this specification with the virtual message processor specification, nodes of any complexity can be specified. An example is shown in Figure 4. In this example, the large circles represent nodes, the smaller grey circles represent virtual message processors, and the arrows represent communications links. The model has three nodes each of which has a different number of virtual message processors and where the communication topology is very specific.

5.4 Modeling the Application

In modeling critical infrastructure systems, it is essential to model the *application functionality* that is necessary for the problem being studied but to omit all unnecessary detail. The range of functionality is considerable, and this presents a challenge in the design of a modeling system. The approach taken in RAPTOR is to allow the functionality of the different nodes in a system to be defined in a high-level language (C++). The advantage of this approach is that it permits any form of functionality to be expressed and any required level of detail achieved. The effort involved is commensurate with the results, however, in that building a model in this context requires some programming effort.

5.5 Modeling Time

A comprehensive notion of *virtual time* is supported in the RAPTOR modeling system. A clock is main-

tained by the system that is incremented when all virtual message processors are either idle because they are blocked waiting for a message or have blocked themselves waiting for time to advance.

The virtual time mechanism permits the modeling of both computational and communication delays throughout any given model. Computational delay is modeled by individual virtual message processors determining that they have completed as much work as they should in a single time interval. In this way, virtual message processors can operate at any relative speed and they can adjust their speed if necessary because self-enforced blocking on time can be conditional.

Communication delay is modeled by defining a delay between the sending of a message and the time when it should arrive. Thus, a virtual message processor can send a message and indicate that its delivery should be delayed by t time units from the time it is sent.

This time mechanism allows studies of performance including throughput, distributions of delays, delays associated with specific events, and so on.

5.6 Modeling Hazards, Threats and Vulnerabilities

Vulnerabilities are circumstances that can lead to the failure of a system. A security vulnerability, for example, is an aspect of a system that an adversary might exploit to harm a system or to steal information. Simple but unexpected vulnerabilities have been the route by which many serious virus attacks have propagated. Vulnerabilities are both common and very complex in critical infrastructure systems. In almost all cases, either the existence of a vulnerability is unknown or the vulnerability is not considered likely to be attacked. This aspect of the problem makes modeling vulnerabilities and their effects at the level of a single node essentially impossible.

The RAPTOR modeling system supports the introduction of known vulnerabilities into nodes as properties that the nodes have. This is done by requiring that the virtual message processor code maintain a data structure for all the vulnerabilities in the model with a parameter reflecting that virtual message processors susceptibility to each vulnerability. The parameter might be absolute (yes or no) or it might be probabilistic for each vulnerability. This mechanism permits, for example, statistical analysis of the effects of vulnerabilities that have certain distributions across the network. As an example, almost all of the nodes in a

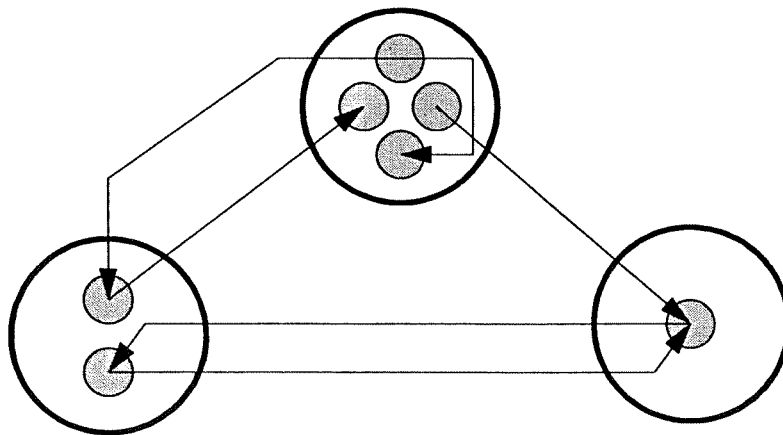


Figure 4. Example Simple Network of Complex Nodes

model might be defined not to be susceptible to a security attack (such as password guessing) but a small fraction, say 0.001%, defined to be susceptible.

Symptom injection is the way in which events such as security attacks and hardware failures are modeled, and the RAPTOR modeling toolset provides a set of fault types, occurrence rates, and durations. Working with the clock that is recording virtual time, the modeling system dispatches symptoms to select nodes or links in any prescribed sequence that is of interest to the researcher. This mechanism is implemented using the message-passing facilities so nodes merely receive a message at a specific virtual time announcing that an event (such as a hardware failure) has occurred. Associated application semantics (such as cessation of all activity within the node) is then effected by the node's "application" software.

Combined with the vulnerability mechanism, symptom injection allows the researcher to model complex fault and attack scenarios. For example, sequences of symptoms can be injected over time (either a short time or a long time) and these symptoms can be taken to represent coordinated security incidents. Similarly, a mixture of faults can be modeled including incidents such as a terrorist attack followed by a coordinated security attack. Each of these has to be programmed as a separate vulnerability and separate symptom sequences but the requisite programming is simple and permits great flexibility.

5.7 Modeling Survivability Mechanisms

Virtual message processors provide network addressing and message transmission at the model level to support required communication structures. They are not, however, limited to being used as elements of an infrastructure model. Thus arbitrary architectural extensions can be modeled using virtual message processors as the building blocks. For example, a sophisticated control-system architecture can be effected simply by merely defining the requisite topology (what nodes there are and to which application-system nodes they will be connected) and by defining the control-system functionality just as the application functionality is defined.

Shell and wrapper architectures are trivial to implement in a RAPTOR model since virtual message processors are responsible for processing messages they receive but there is no system-prescribed processing. Thus a shell can be effected by merely arranging for all incoming or outgoing messages, or both, to be processed by a second set of message processors in addition to those used for basic application functionality within a given virtual message processor.

5.8 Observing Models

Being able to *visualize* what is going on in a critical infrastructure system is important if humans are to be involved in decision making yet the technology for visualizing large networks is not adequate for the very large networks that are at the heart of critical infrastructure systems. The RAPTOR modeling system permits data acquisition of any form and at any time because data acquisition can be programmed as part of the application software. RAPTOR provides a display framework to permit the development of displays suitable for any particular model. The framework provides a collection point for data together with a link to a remote process that can be programmed to include any display features that the model requires. Since the display is remote, it can be executed by a different computer thereby minimizing impact on model execution.

6 Evaluation

The versatility of the virtual message processor construct has provided us the ability to build infrastructure models and to evaluate experimental survivability mechanisms rapidly and easily. We illustrate the use of the modeling system with an example. We have used the RAPTOR system described in this paper to build

several models of a critical application from the banking system, namely the U.S. financial payment system. Our selection of this particular application for modeling was based on our more detailed understanding of the domain—it is arbitrary and of no particular significance. For the example, we describe the largest and most comprehensive model that we have built to date.

The research goal for this particular model is to explore a survivability architecture in which a supplementary hierarchical control system is used to react to widespread failures of several types [3, 14, 20]. This case addresses three critical research issues: (1) the feasibility of distributed, hierarchical control as a survivability architecture; (2) the control algorithms required to respond to failures of different types; and (3) mechanisms that could be employed to specify survivability control policies for the control system.

In addition to these research questions, the development of this model was also used to evaluate the modeling system itself. Clearly issues such as performance, utility, flexibility, and ease of use were of concern.

6.1 Model Architecture

The overall network structure in this model is a tree. This is typical of the way banks are connected for payment purposes, but our topology is strictly hypothetical. The model includes three types of application node. The first is a *branch* bank providing customer service. Such nodes appear as leaf nodes in the tree. The second type of node is a *money center* bank—essentially the primary information center for a single commercial bank. A money-center bank appears as an intermediate-level node in the tree, and has a set of local banks as children. The third type of node represents the *Federal Reserve System* and is the root of the tree. Money-center banks are connected to the Federal Reserve System.

Needless to say, the information system that effects payment in the U.S. is a very large network, and we could not model this exact scale although the model is of the right order. Our current model is composed of almost 10,000 application nodes with almost 100 money-center banks, each of which has 100 associated branches.

The model of the Federal Reserve System includes a primary server and two geographically remote warm spares that permanently mirror the data held by the primary server. They are able to provide service to the remainder of the network if the primary fails, but, in order to do so, the money-center banks must reroute payment requests and wait for service to be initialized. This model is representative of the availability mechanisms actually used by the Federal Reserve System.

6.2 Application Functionality

The application functionality we have implemented in the model includes check processing and large electronic funds transfers. Each payment demand includes typical routing information—source bank, source account number, destination bank, and destination account number as well as the payment amount. User's bank accounts are held at the branch banks and it is there that all payment requests are made. A load generator (another virtual message processor) creates random sequences of payment demands that take the form of either a "check" or an EFT request.

As in the real payment system, payment demands below a certain threshold value are grouped together so that funds transfers between money-center banks are handled in bulk by the Federal Reserve System at scheduled settlement times. Bulk funds received by a money-center bank have to be dispersed through the bank's own network so that the correct value reaches each destination account. This part of the application models the processing of paper checks. Transfers of funds where the value exceeds the threshold value are effected individually and upon receipt of the demand. This aspect of the application models large EFT request processing. The two-tier approach to payment processing is representative of the overall structure

of the real payment system.

6.3 Architectural Supplement

We have described the control systems architecture with which we are experimenting in detail elsewhere [20]. It is a (typically) distributed system that is separate from the application system which senses the state of the application and reconfigures it in the face of adverse circumstances by sending it reconfiguration commands. Sensing the state of application nodes and transmitting commands for reconfiguration takes place conceptually via protection shells that surround application nodes. To permit reconfiguration to be tailored to different semantic levels in the application network topology, a control system typically operates hierarchically with lower levels supplying summary information to upper levels to optimize control decisions.

The function of the control system is to implement *survivability policies*. A survivability policy describes a non-local event of concern (such as a failure of more than a certain number of application nodes), and the recovery commands and their sequence that are have to be sent to application nodes for that particular event. The actual continued service is implemented by the application—it is its invocation that is the responsibility of the control system.

The control system used in the evaluation activity has ten nodes that are connected to groups of application nodes. Each of three of the control-system nodes is connected to one of the Federal Reserve processing nodes together with roughly ten of the money-center banks. The control system observes the application network and determines action based on error detection and error recovery specifications that have been included in the control algorithm.

For purposes of experimentation, the current model implements four responses that can be used for a variety of faults that might arise. These responses are designed to demonstrate and evaluate key aspects of the modeling system and the control-based survivability architecture:

- *Federal Reserve Redirection*
This response requires that the entire payment system switch to the use of a warm spare in the Federal Reserve System.
- *Node Isolation*
This response requires that a node whose intrusion-detection system is triggered be isolated and ignored by the remainder of the network.
- *System-Wide Key Replacement*
This response requires that the entire payment system switch cryptographic keys and account passwords.
- *Comprehensive Shutdown*
This response requires that the entire payment system be shut down.

In a specific scenario that has been used for evaluation, the banking system is first attacked by a terrorist who “bombs” the primary Federal-Reserve server. This results in a switch to the warm spare for continued service. The terrorist bomb is followed by a coordinated security attack in which a series of money-center and branch banks are attacked one after the other. As each attack is detected, the control system directs that the associated node be ignored. After five attacks have been observed, the control system directs a system-wide change of cryptographic keys and account passwords. At that point, communication with nodes that had been attacked and were being ignored is restored. Finally, when a total of ten attacks have taken place, the control system orders the entire payment system to be shutdown.

None of the key services required by transaction processing systems (such as two-phase commit protocols) are provided by the modeling system, nor are they intended to be. The modeling of continued service that is being developed in this example is at the level of system and application management. We are abstracting away important issues such as consistent recovery in distributed heterogeneous systems. Our focus is, instead, on monitoring and control in large distributed systems. We assume that lower level details are provided by the application. They could be added explicitly as part of the application functionality in a model built for a different research goal.

The control system does implement a two-phase commit protocol in this particular modeling exercise and uses it to ensure that consistent decisions are made about what control-system node is to do what regarding recovery.

6.4 Model Implementation

The topology of this model is defined entirely in the topology specification. Application nodes are virtual message processors and the application's communications system is implemented by links between virtual message processors. The control system architecture model is also built with virtual message processors.

The application functionality is implemented by small sections of C++ source providing message interpretation in the application nodes. The functionality implied by the redundancy model for the Federal Reserve System is achieved with a trivial amount of programming within the application functionality.

6.5 Results

Our results to date are in three areas: (1) the utility of the system; (2) the performance of the system; and (3) the feasibility of hierarchic control of network survivability using the control system paradigm. In the first area, the modeling system supported well and in all respects development of the model that we have discussed. Building of the model was easy. Its specification is short. The facilities of the model, especially the pattern of use of components, met all of our demands. We were able to build several versions of the model quickly (in a few days) and incrementally.

To date we have assessed the runtime performance of the system (as opposed to its support for model construction) informally and in only a single area—the runtime performance of a physical computer. On a Pentium-based machine with 64 Mbytes of main memory and a typical disk configuration, acceptable performance is obtained with up to about 10,000 virtual message processors running concurrently processing messages associated with the payment-system model.

Finally, the model described here incorporates a preliminary hierarchic control system that is designed to provide significant survivability enhancement. Although no performance quantification has been undertaken, the model demonstrates the feasibility of network-wide state assessment and damage assessment coupled with a hierarchic approach to state restoration, and continued service. The latter is especially important since survivability of large distributed applications will almost certainly require the following two activities to cope with major failures:

- Significant reconfiguration of the network's topology (state restoration) where different elements of the reconfiguration are coordinated yet tailored to different circumstances throughout the network.
- Substantially different alternative or reduced applications (continued service) at different locations based again on the different circumstances throughout the network.

Using a data collection facility integrated into the model, we measured "transactions" successfully com-

pleted per unit time. Here a transaction is a retail payment, either a “check” or an “EFT” order. Running the model with the survivability enhancement disabled, the transaction rate dropped almost to zero as soon as the Federal Reserve server was bombed. With the survivability mechanism in place the system maintained a reasonable rate of transaction processing with dips occurring as each trauma hit the system.

7 Related Work

The use of models to explore systems that do not admit direct or comprehensive manipulation is not new. Models are used widely, for example, as tools for computer architecture design and development [12, 18, 19, 23] to permit design trade-offs to be studied that would be infeasible by any other means. Simulation is also used, of course, to provide environments that are not otherwise readily available.

There are several existing frameworks and/or toolkits available that support the development of distributed systems. For example, Java’s Remote Method Invocation (RMI) [21] supports a distributed object model which abstracts the communication interface to the level of an object method invocation. For our purposes, however, RMI is not close enough to the semantics of infrastructure applications (message passing) to permit the simple development of models.

The Common Object Request Broker Architecture (CORBA) [17] is a conceptual “software bus” that allows applications to communicate with one another, regardless of who designed them, the platform they are running on, the language they are written in, and where they are executing. The emphasis in CORBA is a distributed system paradigm that promotes interoperability. Again, however, the software that supports CORBA cannot be used easily to build the operational needs that we require.

Much research has addressed the issue of fault tolerance in distributed applications. The Isis toolkit [5], for example, provides a set of mechanisms to support reliable communication in a process group. Other examples include the work on replication of Fabre et al. [9] and transaction models by Chelliah et al. [6]. Existing techniques do not deal with modeling in the sense used with the system that we describe, including transparent insertion of architectural elements, and control and measurement of experimental survivability models. Nor do existing techniques deal with the multitude of system-wide issues that arise in critical infrastructure applications.

Modeling and characterizing of distributed systems have been studied extensively. Andrews discusses process types and process interaction in distributed systems [1]. Nikolaidou et al. describe a Distributed System Simulator (DSS) [15]. The DSS is an integrated environment for performance evaluation of distributed systems. A distributed system is viewed as a combination of a distributed application and a network infrastructure. The DSS permits analysis of the behaviors of the network infrastructure under conditions imposed by the defined distributed application and estimates performance parameters. Though related, our system differs in that the objective is to investigate vulnerabilities of new and existing infrastructure systems and to study ways to deal with them, rather than addressing traditional performance issues.

8 Conclusions

Dealing with the survivability issues posed by modern critical information systems in infrastructure applications presents many challenges. The systems are large, usually depend upon COTS components, contain extensive legacy code, and must meet multiple diverse dependability requirements. The need for improved survivability is increasing as more and larger systems are deployed, as society becomes more dependent on critical infrastructures, and as some threats (such as the possibility of national-scale malicious attacks) become more likely and their perpetrators more sophisticated.

A serious impediment to research in this field is the difficulty of experimentation. Real systems cannot be the subject of experiment for the most part because real systems are, by definition, critical and their operators cannot risk the possibility of damage during experimentation. We have described a modeling system that begins to address this problem by supporting the development and evaluation of operational models of infrastructure applications and survivability mechanisms.

The system that we have described allows us to develop executable models very quickly. Preliminary results based on an example model have shown that the system meets the basic requirements set forth, and that model runtime performance is adequate for our experimental purposes. The role of the system in our research is to permit experimentation with architectural concepts that support survivability. The model that we have described demonstrated a hierarchic control mechanism providing error detection, damage assessment, and facilities for continued provision of service that can be tailored to the needs of different parts of a specific application. Of course, we have not established the utility of such a control system for real infrastructure survivability. Nevertheless, our modeling system and research method provide a basis for undertaking the basic experimental research needed to begin to evaluate such concepts.

Acknowledgments

This effort sponsored in part by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0314. The U.S. Government is authorized to reproduce and distribute reprints for governmental purpose notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government. Kevin Sullivan was also supported by the National Science Foundation under grants CCR-9502029, CCR-9506779, CCR-9804078.

References

1. G. R. Andrews, Paradigms for Process Interaction in Distributed Programs, *ACM Computing Surveys*, Vol. 23, No. 1, March 1991, pp. 49-90.
2. R. L. Bagrodia and C.-C. Shen, MIDAS: Integrated Design and Simulation of Distributed Systems. *IEEE Transactions on Software Engineering*, Vol. 17, No. 10, 1991, pp. 1042-1058.
3. R. Bateson, *Introduction to Control System Technology*, (6th ed.), Upper Saddle River, NJ: Prentice Hall, (1998).
4. M. A. Bauer, R. B. Bunt, A. El Rayess, P. J. Finnigan, T. Kunz, H. L. Lutfiyya, A. D. Marshall, P. Martin, G. M. Oster, W. Powley, J. Rolia, D. Taylor, and M. Woodside, Services Supporting Management of Distributed Applications and Systems, *IBM Systems Journal*, 36, 4, (1997), 508-526.
5. K. P. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, 1994.
6. M. Chelliah and M. Ahamad, *Multi-Model Fault-Tolerant Programming in Distributed Object-Based Systems*. Technical Report, GIT-CC93/72. College of Computing, Georgia Institute of Technology.
7. Defense Modeling & Simulation Office (DMSO), Introduction to the High Level Architecture. Simulation Interoperability Workshops, September 8-12, 1997. See <http://hla.dmsomil/hla>.
8. R. J. Ellison, D. A. Fisher, R. C. Linger, H. F. Lipson, T. Longstaff, and N. R. Mead, *Survivable Network Systems: An Emerging Discipline*, Technical Report CMU/SEI-97-TR-013, Software Engineering Institute, Carnegie Mellon University, (November 1997).
9. J.-C. Fabre, V. Nicomette, T. Perennou, R. J. Stroud, and Z. Wu, *Implementing Fault Tolerant Applications using*

- Reflective Object-Oriented Programming*. Technical Report, LAAS-CNRS, LAAS-CNRS#94156, March 1995.
10. J. C. Knight, M. C. Elder, J. Flinn, and P. Marx, *Summaries of Three Critical Infrastructure Applications*, Technical Report CS-97-27, Department of Computer Science, University of Virginia, Charlottesville, VA 22903 (December 1997).
 11. J.C. Knight, R. W. Lubinsky, J. McHugh, and K. J. Sullivan, *Architectural Approaches to Information Survivability*, Technical Report CS-97-25, Department of Computer Science, University of Virginia, Charlottesville, VA 22903 (September 1997).
 12. S. F. Lundstrom and M. J. Flynn, *Design of Testbed and Emulation Tools*, Technical Report, CSL-86-309, Computer Systems Laboratory, Stanford University, Sept. 1986.
 13. K. Marzullo, R. Cooper, M. D. Wood, and K. P. Birman, Tools for Distributed Application Management, *IEEE Computer*, (August 1991), 42-51.
 14. R. Murray-Smith and T. A. Johansen (eds.), *Multiple Model Approaches to Modeling and Control*, Taylor & Francis: London, UK, (1997).
 15. M. Nikolaidou, D. Anagnostopoulos, and P. Georgiadis, *Modeling and Simulation of Distributed Systems*, Technical Report TR97-0011, Department of Information, University of Athens, Greece.
 16. Office of the Undersecretary of Defense for Acquisition & Technology, Report of the Defense Science Board Task Force on Information Warfare-Defense (IW-D), (November 1996).
 17. A. Pope, *The COBRA Reference Guide: Understanding the Common Object Request Broker Architecture*, Addison Wesley, 1998.
 18. M. Rosenblum, S. A. Herrod, E. Witchet, and A. Gupta, Complete Computer Simulation: The SimOS Approach, *IEEE Parallel and Distributed Technology*, Fall 1995.
 19. C. Ruemmler and J. Wilkes, An Introduction to Disk Drive Modeling. *IEEE Computer*, March 1994, pp. 17-28.
 20. K.J. Sullivan, J.C. Knight, X. Du and S. Geist, Information survivability: a control systems perspective, *Proceedings of the 1999 International Conference on Software Engineering*, Los Angeles, May 1999.
 21. Sun Microsystems, *Remote Method Invocation Specification*. 1997. <http://java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html>.
 22. United States Government Printing Office (GPO), No. 040-000-00699-1, *Protecting America's Infrastructures: Report of the Presidential Commission on Critical Infrastructure Protection* (October 1997).
 23. J. E. Veenstra and R. J. Fowler, MINT: A Front End for Efficient Simulation of Shared Memory Multiprocessors, *Proceedings of the Second International Workshop on Modeling, analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 1994, pp. 201-207.