Introduction and Overview

Information System Survivability

Peter G. Neumann

University of Maryland ENPM 808s

Fall Semester 1999

ENPM 808s:
Information Systems Survivability:
1. Introduction and Overview

\- -

Introduction to INFOSURV: overview, outline of the course, expectations, guidelines for the course project, concepts, compromisibility, defenses

The URL for this document provides the lecture slides and pointers to the course materials. This collection of material will grow lecture by lecture during the course, and should be checked before each lecture for up-to-date versions. The html version is generated by Otfried Cheong's Hyperlatex program. I have intentionally not kept the entire set of slides in one file because the materials for each lecture will grow as the course progresses. (Caution: The numbering of the files is off-by-one from the numbering of the lectures, but html does not care.)

Students should acquire the requisite background reading materials in advance of the course, and check this file weekly during the course.

The 14 sections of lecture materials are oriented roughly toward each of the 14 class periods. However, the course may move ahead or lag behind, depending on the actual presentation of the material, and thus students should anticipate that the lectures may not stick strictly to the section boundaries outlined here. [Added note: We came close, except for the third class period being blown away by Hurricane Floyd, and being merged into the fourth class period.]

The reading material is intentionally front-loaded, leaving time free for the final project during the last half of the course.

The Nature of the Course

\- -

In many respects, this is an unconventional course. Although it has a strong engineering perspective, it also has aspects related to computer science, economics, philosophy, ethics, and human well-being. It has an unusually pervasive system sense, with emphasis on principles and experience rather than on quick-and-dirty would-be solutions that in reality are inadequate. It has a component of computer-system history, in that it recapitulates significant earlier achievements and lessons that seem to have been largely forgotten.

The subject matter of attaining high survivability inherently does not lend itself to cookbook approaches. You will not be expected to memorize and regurgitate large amounts of information. You will be expected to think, reason, generalize, and draw your own conclusions.

There will be no final exam or homework (other than reading and thinking). Instead, there will be a single final project. In that there is generally no single set of correct answers, your work is intended to challenge your creativity and imagination rather than requiring rote responses.

University of Maryland ENPM 808s
Fall Semester 1999
Thursday Evenings, 7:00–9:40p.m. ET

- -

Peter G. Neumann
Computer Science Laboratory
SRI International
Menlo Park, CA 94025-3493
Neumann@CSL.sri.com
1-650-859-2375
http://www.csl.sri.com/neumann/
On-line course materials can be found at
http://www.csl.sri.com/neumann/umd.html
Course slides may differ from on-line files.

Course Materials Most of the course materials are or will be on-line.

- -

1. Schedule of lectures, background:
http://www.csl.sri.com/neumann/umd808s.html

2. The general textbook for the course:
Peter G. Neumann, *Computer-Related Risks*, Addison-Wesley, ISBN 0-201-55805-X.

3. More recent further material:
http://www.csl.sri.com/neumann/risks-new.html

4. *Practical Architectures for Survivable Systems and Networks* is browsable:
http://www.csl.sri.com/neumann/arl-one.html
also available in PostScript and pdf forms:
arl-one.ps and arl-one.pdf
The report contains many relevant references, including those cited here.

5. *Illustrative Risks* provides a frequently updated topical abstract index to the Risks Forum material:
http://www.csl.sri.com/neumann/illustrative.html
as well as
http://www.csl.sri.com/neumann/illustrative.ps
http://www.csl.sri.com/neumann/illustrative.pdf

Reuse of the Course Materials

- -

Corrections, suggestions, and new materials appropriate for the teaching of this course would be most welcome. Unless otherwise specified, it will be assumed that your contributions may be included herein or linked to your own site (immediately, or perhaps in any subsequent manifestations of this course) with explicit credit given to the contributors and with essentially the same open reuse rights as the original body of on-line course materials.

Acknowledgements, Disclaimers

- -

The work on which this course is based is supported by the U.S. Army Research Lab under Contract DAKF11-97-C-0020. LTC Paul Walczak (PWalczak@arl.mil), 1-301-394-3862 is the contact.

All opinions, findings, conclusions, and recommendations are mine and do not necessarily reflect the views of ARL, or SRI International, or the University of Maryland.

Topics To Be Covered

Illustrative risks that motivate the needs for survivability

Analysis of survivability requirements and their interdependencies, with survivability explicitly dependent on security, reliability, performance, etc.

Identification of inadequacies in existing commercial systems, missing components, and approaches and architectures for overcoming those inadequacies

Explicit conceptual subsystem designs for the missing components, with recommendations for their implementation

Survivable architectural structures that facilitate subsystem integration, including securely integrated cryptography

Establishment of structural system and network architectures that can achieve the desired overall survivability, including robust mobile code and systems that reduce trust necessary in end-user systems

Recommendations for future research and development, education, training, etc.

Basic Course Outline

```
1.   2 Sep Introduction and overview
2.   9 Sep Survivability-related risks
3.  16 Sep Risks continued, and Threats
4.  23 Sep Survivability requirements
5.  30 Sep Deficiencies in existing systems
6.   7 Oct Overcoming these deficiencies 1
7.  14 Oct Overcoming these deficiencies 2
8.  21 Oct Architectures for survivability 1
9.  28 Oct Architectures for survivability 2
10.  4 Nov Reliability in perspective
11. 11 Nov Security in perspective
12. 18 Nov Architectures for survivability 3
--  25 Nov Thanksgiving.  No class.
13.  2 Dec Implementing for survivability
14.  9 Dec Conclusions
```


Classes

Classes will generally consist of about an hour and fifteen minutes (lecture-style, with some questions permitted at UMd), a break, and then informal discussion driven to a considerable extent by student questions for the remainder of the class period (with possibly some remote questions where first-level satellited video facilities make that feasible).

Projects

In lieu of a final exam, a class project will be due at the last class, on 9 December. A few suggestions for possible topics are at http://www.csl.sri.com/neumann/umd.html. Proposals for projects are due by e-mail no later than 22 October, with responses of acceptances or required iteration to be available by e-mail prior generally about a week later. Earlier submissions will allow for more iteration if needed and more time for your project. Projects should be scoped not to exceed the normal load of homework, studies, and final exam in a comparable course. Group projects may be proposed, although they should clearly reflect who is going to do which portions of the total effort.

Illustrative Project Topics

A report analyzing a relevant topic, such as robust algorithms, authentication, preventing service denial, roles of public-key cryptography, how to robustify open-source components

A paper design of a survivable subsystem or component and how it might integrate into a robust entirety.

A robustification effort based on existing open-source software, either as a paper or as a software development.

An analysis of how to eliminate or mitigate a common software flaw (e.g., buffer overflow): via programming languages, precompiler, style or system constraints, misuse+anomaly detection, etc.

A “challenge” from *Computer-Related Risks*

Survivability

- -

In the present context, survivability is the ability of a computer-communication system-based application to satisfy in an ongoing basis certain specified critical requirements (for example, security, reliability, real-time responsiveness, and correctness), in the face of adverse conditions. In some cases, survivability may require reconfigurability, interoperability, etc.

Anticipated adversities might typically include hardware faults, software flaws, attacks on systems and networks perpetrated by malicious users, accidental misuse, environmental hazards, unfortunate animal behaviors, acts of God, etc.

Survivability Challenges

- -

System and network survivability ultimately depends on the reliability, fault tolerance, security, performance, and operational robustness of many constituent systems and networks.

Operational survivability depends on hardware, software, power, communications, environmental factors such as electromagnetic interference, extreme weather, earthquakes, etc.

Operational survivability also depends on the knowledge, training, and integrity of many people (procurement officers, system developers, administrators, users, and even innocent bystanders) and the dynamic ability of system administrators to perform emergency maintenance

Infrastructures

- -

Critical national infrastructures: telecommunications, power and energy, water, transportation, banking and finance, emergency services, continuity of government, etc.
See http://www.pccip.gov.

Information infrastructures: the Internet and private nets

Computer-communication systems: operating systems, database management systems, networking software, ...

Difficulties To Be Overcome

- -

There are generally no easy answers. Cookbook approaches are insufficient.

Pervasive understanding of the fundamentals is essential, not just a scattering of so-called expertise.

Survivability is an overarching *emergent* property. That means it is a property of a system or network in the large, not just a property of its components that can be merely stuck together. Consequently, survivability of a whole system cannot be analyzed locally. It depends on many factors (discussed below).

Composition of subsystems is often unpredictable, with unanticipated side-effects.

Survivability is not easily retrofitted onto nonsurvivable applications.

System survivability is often inadequate in existing systems, which are terribly deficient (for example, in security and reliability)

Basic Needs

- -

Definitive survivability requirements and subrequirements

Generic system/network architectures

Robust protocols and open-system components

Realistic operational prototypes with nontrivial survivability characteristics

Fully fledged robust commercially supported systems, nonproprietary where possible

Systematic use of good cryptography for authentication, integrity, and confidentiality

Applicable research and development, well integrated into robust systems

Concepts

Noncompromisibility from outside, from within, from below, in the presence of all survivability-relevant threats

Trustworthiness, dependability, assurance

Generalized composition, encompassing all useful modes of subsystem composition

Generalized dependence, compensating for and overcoming underlying untrustworthiness

Generalized survivability, based on many generalized-dependence mechanisms

Compromises

Compromise from outside: from above, or laterally at the same layer (e.g., access with no authorization, exploiting a logic flaw). Letter bombs; spoofing; penetrations

Compromise from within: using privileges of the given layer. Programming errors; internal Trojan horses; authorized misuse

Compromise from below: from a lower layer (e.g., hardware-based attacks on software, OS-based attacks on applications). Sniffers; Ken Thompson's C Compiler Trojan horse; hardware flaws or alterations

Illustrative Compromises

Layer of abstraction	Compromise from outside; Needs exogirding	Compromise from within; Needs endogirding
Outside environment		Acts of God, earthquakes, lightning, etc.
User	Masqueraders	Accidental mistakes Intentional misuse
Application	Penetrations of application integrity	Programming errors in application code
Middleware	Penetration of Web and DBMS servers	Trojan horsing of Web and DBMS servers
Networking	Penetration of routers, firewalls Denials of service	Trojan horsing of network software
Operating system	Penetrations of OS by unauthorized users	Flawed OS software Trojan-horsed OS Tampering by privileged processes
Hardware	Externally generated electromagnetic or other interference External power-utility glitches	Bad hardware design and implementation Hardware Trojan horses Unrecoverable faults Internal interference
Inside environment	Malicious or accidental acts	Internal power supplies, tripped breakers, UPS/battery failures

Illustrative Compromises, continued

Layer of abstraction	Compromise from within; Needs endogirding	Compromise from below; Needs undergirding
Outside environment	Acts of God, earthquakes, lightning, etc.	Chernobyl-like disasters caused by users or operators
User	Accidental mistakes Intentional misuse	Application system outage or service denial
Application	Programming errors in application code	Application (e.g., DBMS) undermined within operating systems (OSs)
Middleware	Trojan horsing of Web and DBMS servers	Subversion of middleware from OS or network operations
Networking	Trojan horsing of network software	Capture of crypto keys within the OS Exploitation of lower protocol layers
Operating system	Flawed OS software Trojan-horsed OS Tampering by privileged processes	OS undermined from within hardware: faults exceeding fault tolerance; hardware flaws or sabotage
Hardware	Bad hardware design and implementation Hardware Trojan horses Unrecoverable faults Internal interference	Internal power irregularities
Inside environment	Internal power supplies, tripped breakers, UPS/battery failures	

Trust and Trustworthiness

- -

Trust and Trustworthiness are different.

Some (sub)systems are more trustworthy with respect to certain threats (to security, reliability, etc.) because their design and implementation reduce the likelihood of being compromised in those respects.

Dependability is a perception of trustworthiness associated with a specific requirement

Assurance is a measure of faith that can be placed in trustworthiness and dependability.

Some (sub)systems are trusted to behave properly even when they are untrustworthy. This is a very serious problem. Misplaced trust can lead to compromises.

Generalized Composition

- -

Composition must accommodate serial connections, feedback, hierarchical layering, networking, collateral dependence, mutual suspicion, clients/servers, guards, ... without undermining the given requirements and component interoperability

Many commercial systems are not easily composed with other systems.

Much theoretical research on composition is not applicable to realistic systems. TCSEC criteria are deficient (Orange Book, Red Book).

Trustworthiness depends critically on the noncompromisibility of compositions. Compositions may be untrustworthy despite local trustworthiness of the components.

Generalized Dependence

- -

Strict dependence (the hierarchical "uses" relation of Parnas 1974) demands strict correctness of underlying mechanisms to attain correctness of a given mechanism.

Generalized dependence enables acceptable operation despite faults, failures, errors, and misbehavior or misuse of underlying mechanisms. It compensates for, bypasses, or otherwise overcomes errant behavior. Thus, it enables the enhancement of trustworthiness.

Generalized dependence approaches the problem of trying to "make a silk purse out of a sow's ear." It can succeed quite effectively in certain respects in various cases (as considered next).

Identified Types of Generalized Dependence for Enhancing Trustworthiness (See Section 1.2.5 of http://www.csl.sri.com/neumann/arl-one.html.)

- -

Reliability: error-correcting coding, Moore-Shannon and von Neumann theories, Byzantine algorithms, self-resynchronization, and fault-tolerance techniques generally

Security: Domains with mutual suspicion, trustworthy firewalls, guards, and wrappers, crypto and especially multikey or Byzantine crypto, real-time analysis and response, but not necessarily kernels and TCBs, which assume bottom-up trustworthiness

Survivability: Run-time checks on all requirements, underlying security and reliability as well as survivability, real-time analysis extended to survivability

Generalized Dependence: Error-Correcting Codes

- -

Claude Shannon long ago showed how to signal through a noisy channel with arbitrarily high reliability by using sufficient redundancy.

The field of error-correcting codes has evolved considerably since then, with codes designed to detect and/or to correct certain patterns of errors: random, asymmetric (e.g., 1-to-0 only, or 0-to-1 only), bursty, or otherwise correlated, in block, variable-length, and sequential communications, as long as the required redundancy does not cause the available channel capacity to be exceeded.

The best known and simplest class of error-correcting codes is the Hamming Code, which can correct any random single-bit error in a code word of length n, in which k bits are information bits, and $n-k$ bits are redundant check bits, where $n = 2^{n-k} - 1$; in principle, the $n-k$ redundant bits provide $2^{n-k} - 1$ error syndromes pinpointing where the (single) error occurred, and an all-zero syndrome denotes the case of no error. (More later.)

Generalized Dependence: Moore and Shannon 1956: “Crummy Relays”

- -

Given a box of relays of unknown goodness, test them for reliability. Throw out those with failure probability near one-half. Accept those with lesser failure probability, using positive logic. Reverse the logic on those with greater failure probability so that they then be considered to have failure probability less than one-half. As an oversimplified example, suppose each relay has failure probability p less than one-half when it is required to provide a closed path. Then, because only one good path is sufficient to close the circuit, a parallel circuit of n logically equivalent relays with identical failure probabilities has an aggregate failure probability p^n, which can be arbitrarily small for large enough n. Then, use serial-parallel combinations to attain arbitrarily low failure probabilities.

Generalized Dependence: Byzantine Agreement

- -

Whereas error-correcting codes make some assumptions about the transmission or storage medium and about the worst-case failures, and Moore-Shannon circuits make assumptions about local failure probabilities, Byzantine agreement makes no local assumptions about any components. Each component that fails may fail in any arbitrarily nasty way.

For example, a Byzantine clock (Lamport et al. 1982) with $3k+1$ subclocks can tolerate arbitrary failure modes in any k clocks, including malicious modes. (This cannot be done with fewer clocks unless constraining assumptions are made.)

Generalized Dependence for Security: Guards, Firewalls, and Wrappers

- -

A security guard might try to prevent the outflow of sensitive information and prevent the inflow of Trojan horses that could inflict damage on the guarded system.

A firewall might try to prevent unauthorized users from accessing an internal system from outside, and prevent certain types of access as well. Unfortunately, some types of traffic (e.g., http) can compromise internal security.

A wrapper might try to mediate access to a flawed component and to block harmful use either on the way in or the way out.

In principle, these components attempt to overcome the shortcomings of the systems that they supposedly protect, in the spirit of generalized dependence.

Today’s guards, firewalls, and wrappers tend to be incomplete in their requirements, flawed, and often bypassable, penetrable, and compromisible from within and below. “Trusted paths” to PCs are a challenge.

Even if they were more trustworthy, there would still be risks of compromise.

Generalized Dependence for Security: Protection Domains

- -

In principle, it should be possible to confine the results of execution within a domain, strictly confining any would-be side-effects. This would be particularly valuable with mobile code.

In practice, computing enclosures tend to leak profusely, because of weak hardware and poor software security.

Domains are an old idea, going back to Multics rings in 1965, Mike Schroeder’s 1972 thesis, and capability architectures. Unfortunately, they are seldom used to realize their potential.

Security Instances of Generalized Dependence: Cryptography

- -

Cryptography provides the ability to use a completely unprotected storage or communications medium for sensitive information, to establish integrity of otherwise potentially untrustworthy code and data, and to authenticate the identity of otherwise potentially unknown entities.

Cryptography is almost always vulnerable to compromise from within and below (and denial of service attacks). Compromises are usually easier to perpetrate than brute-force exhaustive cracking of keys. However, 56-bit DES keys can now be broken (e.g., Deep Crack) in a few days with a single computer system, and much more rapidly with multi-system approaches (e.g., Distributed Crack).

Commonality among Generalized Dependence

- -

In all these examples, there are of course pitfalls, including risks of subversion and limitations of the theory when applied to practice. For example, single-error-correcting codes can miscorrect multiple errors. Byzantine systems fail if there are more than k out of $3k+1$ misbehaving systems. Firewalls can be penetrated if they are configured to permit certain traffic through. Wrappers and almost everything else can be compromised from within or from below.

Exploiting Generalized Dependence

- -

Survivability *depends* (in the sense of generalized dependence) on security, reliability, fault-tolerance, performance, ...

Attaining survivability requires a combination of many techniques for ensuring necessary lower-layer properties, including relevant generalized-dependence mechanisms.

In addition, real-time monitoring with detection of survivability-relevant anomalies, automated diagnosis, the ability to correlate across multiple platforms, and stable reconfiguration are survivability-relevant instances of generalized dependence.

Mandatory Policies (introduction)

- -

In principle, mandatory policies have the potential advantage that, once they are established, they cannot be compromised from above. They are very valuable as architectural concepts, although there are many practical problems (as we shall see).

MLS: Multilevel security (Bell and LaPadula)

MLI: Multilevel integrity (Biba)

MLA: Multilevel availability (Neumann-Proctor-Lunt; conceptual)

MLX: Multilevel survivability (new concept, primarily conceptual)

These are considered as architectural concepts later on.

Reading for the Next Class Period

- -

Read Chapter 1 of *Computer-Related Risks.* Pick some of the topics that interest you most in Chapters 2 and 3, and read at least the chapter sections that seem most relevant to survivability.

Consider the Illustrative Risks document on-line at
http://www.csl.sri.com/neumann/illustrative.html
(or .ps or .pdf) and think about how many of those risks cases are survivability related. (Note: The descriptor symbol V is used to denote cases that are particularly surViVability related, although many other cases are also survivability relevant.)

Survivability-Related Risks

ENPM 808s
Information Systems Survivability:
2. Survivability-Related Risks

- -

Survivability-related risks, causes, effects, analysis of case histories, common modalities, lessons to be learned (Y2K, development fiascoes, etc.)

Source materials: *Computer-Related Risks* and the Illustrative Risks summary
http://www.csl.sri.com/neumann/illustrative.html
(or .ps or .pdf)

Considerable background for this class period is found in Chapters 2 and 3 of *Computer-Related Risks,* in the archives of the ACM Forum on Risks to the Public in the Use of Computers and Related Technologies, and summarized in one-liners in the the Illustrative Risks document on-line http://www.csl.sri.com/neumann/illustrative.html (or .ps or .pdf) referring to many of those cases.

Practical Realities

- -

Enormous risks exist in all of the critical national infrastructures, telecommunications, electric power, pipelines, transportation, etc. See the President's Commission on Critical Infrastructure Protection (http://www.pccip.gov) These infrastructures all depend critically on each other, and on information infrastructures.

The U.S. Government and the Dept. of Defense are increasingly dependent on off-the-shelf systems. Those systems are not adequately reliable, available, or secure, not adequately interoperable, reusable, maintainable or evolvable, and consequently not adequately survivable.

Survivable systems and networks require extraordinary human care. Their requirements vastly exceed what is found in practice.

Serious security and reliability vulnerabilities abound in existing systems. Potential threats are dramatically outstripping capabilities. The risks are innumerable. Systems and networks are also very difficult to maintain without introducing new vulnerabilities.

System development and procurement processes are abysmal. Consider some recent fiascoes: the ADATS tank-based anti-helicopter missile system, the Air Traffic Control modernization, the IRS Tax Systems modernization, the FBI fingerprint system and NCIC-2000 development, the impending Y2K crisis, etc.

System operational experience is abysmal. Consider the Yorktown dead in the water for almost three hours due to a divide-by-zero in a Windows NT application; AEGIS software problems; etc.

Significant advances in research and development relating to survivability are very slow in finding their way into the commercial marketplace, and into adoption by the critical-infrastructure providers.

Some Large-Scale Network Outages

- -

1980 ARPAnet collapse. Hardware omission (no memory parity), flaky garbage collection, two hardware errors, propagated to entire net: four-hour outage.

Internet service outages. 1997 black hole due to routing table error: 50,000 addresses pointing to one site. 1986 Northeast disconnect.

Network Solutions Internet domain table error blocks .com and .net for four hours.

1990 AT&T long-distance calling blockage due to SS7 software flaw: crashes propagated incessantly for 11 hours.

1998 AT&T packet-data frame-relay network collapsed US-wide: faulty software upgrade gave 26-hour black-hole effect.

More Large-Scale Case Histories

- -

Summer 1996 Western power outages: 2 July 1996 over 10 Western states (tree, operator, propagation). 10 August 1996 affected 8 million customers in 8 states, Canada, Baja.

May 1998 Galaxy IV satellite failure, primary and backup systems both failed, disabling 40M pager systems, etc. Reconfiguration problematic, moving satellites or antennas, reprogramming, ...

1997 Kansas City power outage triggered national air-traffic snarl: operator shutdown wrong system half in preventive maintenance. Many other air-traffic propagation effects.

Military Operational Risk Cases

- -

Vincennes' AEGIS shootdown of Iranian Airbus

Stark operation against Iraqi Exocets

Recurring Black Hawk problems: EMI, hydraulic failures, friendly fire over Iraq

Patriot missile system clock drift

Sgt York gun software problems

Ships out of commission: USS Hue City and USS Vicksburg software integration problems, plus USS Yorktown noted above

Tomahawk missile test abort

Some Localized Case Histories

- -

Soviet Phobos I and Phobos II failures (misguided ground control)

1984 TDRS outage (cosmic rays)

1986 Challenger disaster (cold O-ring)

Skylab orbit alteration (sunspots)

Atlas-Centaur program alteration (lightning)

emh3.arl.mil 4-day delay on incoming e-mail 24 Sept 1998; dockmaster.nscs.mil loss of week's worth of e-mail in Feb 1998 upgrade.

More Accidental Denials of Service

- -

Telephone system and power outages due to cable cuts, rodents, hardware, software, inadvertent battery usage, Hillsdale fire, floods, air conditioning failures, ...

Collapses of air- and auto-traffic control systems

ATM, bank, credit-card system outages, business interruptions

Electromagnetic interference: Sputnik, Reagan's AF-One, Ft.Detrich (garage-door openers!); MtDiablo/Navy; Quebec power; nuclear reactor; effects on flight controls?

eBay's three-time major losses of service, hardware, software, net connection (after creating hot standby, which was useless in the absence of the net connection!)

See the *Illustrative Risks* compendium for a summary of many cases involving failures of survivability, security, reliability, safety, and human well-being.

A Few Cases of Sabotage

- -

1987 Australian communications blackout (24 cables severed in 10 locations)

1997 San Francisco blackout (switches thrown) (RISKS-19.42)

Atlas rocket database tampering (RISKS-11.95, 12.60); malicious password changes (e.g., Washington D.C. city computer blocked), etc.

Many other cases in *Illustrative Risks,* Trojan horses, logic bombs, time bombs, viruses, Encyclopedia Brittanica, Italian newspaper, Arts Assets, election frauds, ...

Hacked and Altered Web Sites

- -

Justice Department (RISKS-18.35)
CIA (RISKS-18.49)
Air Force (RISKS-18.64)
NASA (RISKS-18.88)
Three Army Web sites (RISKS-19.63)
FBI, U.S. Senate, Department of the Interior (RISKS-20.43)
Other cases in *Illustrative Risks*
The ease with which this can be done is really absurd!

But the lessons being drawn are evidently the wrong ones (e.g., arrest more hackers). The real lesson should be that the computer-communication infrastructure is very weak.

Sources of Risk

- -

The system development process is full of risks in conceptualization, requirements, models, design, implementation, support systems, testing and verification, evolution and maintenance, retrofitting, ... [Problems here are considered in greater detail in the class period on deficiencies.]

System operation and use are also risky: Hardware malfunctions, power failures, environmental factors, software flaws, human (mis)behavior, ...

Proprietary code is often enormous in size, inscrutable in not being subject to outside analysis, inflexible in the unavailability of source code, and likely to force future systems into noninteroperable standards.

People are always an enormous source of risks.

Causes and Effects

- -

Reliability and security are often closely interrelated.

Survivability, like reliability and security, is a weak-link problem. Even with defensive design, there may be weak links or combinations of weak links.

Similar effects may result from either reliability problems, or security attacks, or both: 1980 ARPAnet collapse, 1986 New England separation from the ARPAnet, 1990 AT&T long-distance collapse, electromagnetic interference.

Known potential reliability problems create ideal opportunities for attacks, e.g., at midnight between 31 Dec 1999 and 1 Jan 2000. It may be very difficult to diagnose the causes.

Discussion Topics: Related Risks

- -

Preliminary consideration of lessons to be learned from the survivability-related risks

Do other risks that we have not considered (e.g., human safety, privacy, person well-being, financial fraud, medical health, etc.) have lessons for us that are relevant to survivability? Are they more of the same, or are there real differences?

Reading for the Next Class Period

- -

Read Chapter 2 (survivability threats) of the arl-one report http://www.csl.sri.com/neuma one.html.

Read Chapter 4 of *Computer-Related Risks*. (It is very short.)

Note: The second lecture material on risks is likely to continue somewhat into the third class period.

Threats to Survivability

ENPM 808s
Information Systems Survivability:
3. Threats

- -

Threats to Survivability, Security, Reliability, Predictability, etc.

Wide Range of Illustrative Threats

- -

Inappropriate requirements: wrong, deficient, overly specific, underly constraining, sometimes poor conceptualization

Flawed designs: inconsistent with requirements, new vulnerabilities not covered by requirements

Faulty implementation: inconsistent with design specs, new bugs not covered by specifications, Trojan horses, incomplete testing, short-sighted optimization,

Operation: intentional and accidental misuse by users, system administrators, insiders, outsiders, faulty maintenance, premature decommission

Hardware malfunctions, power failures, environmental disturbances

Multidimensional Threat Domains

- -

Security attacks, malicious or otherwise: See Figure 2.1 and Table 2.1 of the ARL report (which follows). External misuse, hardware misuse, masquerading, pest programs, bypasses of authorization, misuse with authorization, indirect misuse. ...

Reliability failures: See Table 2.2 of the ARL report (which follows). Application code, middleware, DBMS, networks, operating systems, software development, hardware, national infrastructures, environments, interference, ...

Performance failures: partial outages, resource saturation, degradation; may be induced by reliability and security failures, ...

Multiple Threats

- -

Weak links are already riskful. Exploitations of multiple weak-links can be even more far-reaching.

Independent vs. coordinated threats

Sequential threats vs. concurrent threats

Accidental vs. intentional threats

Crossover threats involving security, reliability, etc.

Types of Computer Misuse

- -

External misuse (EX)
1. Visual spying: observation of keystrokes or screens
2. Misrepresentation: deception of operators and users
3. Physical scavenging: dumpster-diving for printout

Hardware misuse (HW)
4. Logical scavenging: examining discarded or stolen media
5. Eavesdropping: electronic or other data interception
6. Interference: electronic or other jamming
7. Physical attack on, or modification of, equipment or power
8. Physical removal of equipment and storage media

Masquerading (MQ)
9. Impersonation: false identity external to computer systems
10. Piggybacking attacks on communication lines, workstations
11. Playback and spoofing attacks
12. Network weaving to mask physical whereabouts or routing

Pest programs (PP): setting up opportunities for further misuse
13. Trojan-horse attacks (including letter bombs)
14. Logic bombs (a form of Trojan horse, including time bombs)
15. Malevolent worm attacks, acquiring distributed resources
16. Virus attacks, attaching to programs and replicating

Bypassing authentication or authorization (BY)
17. Trapdoor attacks, from any of a variety of sources:
a. Improper identification and authentication
b. Improper initialization or allocation
c. Improper termination or deallocation
d. Improper runtime validation
e. Naming flaws, confusions, and aliases
f. Improper encapsulation: exposed implementation detail
g. Asynchronous flaws: e.g., time-of-check to time-of-use anomalies
h. Other logic errors
18. Authorization attacks, for example, password cracking, token hacking

Active misuse of authority (AM) (writing, using, with apparent authorization)
19. Creation, modification, use, service denials (includes false data entry)
20. Incremental attacks (e.g., salami attacks)
21. Denials of service (including saturation attacks)
Passive misuse of authority (reading, with apparent authorization) (PM)
22. Browsing randomly or searching for particular characteristics
23. Inference and aggregation (especially in databases), traffic analysis
24. Covert channel exploitation and other data leakage
25. **Misuse through inaction** (IM): willful neglect, errors of omission
26. **Use as an indirect aid for subsequent misuse** (IN): off-line preencryptive matching, factoring large numbers, autodialer scanning.

Illustrative Reliability Threats

- -

Outside-environmental threats
Environmental problems (earthquakes, floods)
Power utility disturbances
Electromagnetic/other external interference
Inappropriate user behavior, unavailability of key persons
National-infrastructure threats
Glitches in telecommunications, air-traffic control, power distribution, and other infrastructures dependent
on computer-communication infrastructures
Middleware and application-code threats
Windows environments: cache management, crashes
Browser and Web server flaws
Accidentally corrupted code

Database-specific threats
DBMS software flaws
Internal database synchronization and cache management
Distributed database consistency
Improper DBMS software upgrades and maintenance
Improper database entries and updates
Network threats
Faulty network components (hosts, routers, firewalls, etc.)
Distributed system synchronization
Traffic blockage and congestion
Operating-system threats
OS software design and implementation flaws
Improper OS configuration
Improper OS upgrades and maintenance
Failures of backup and retrieval mechanisms

Software-development problems
Faulty system design and implementation,
Poor use of software engineering techniques
Bad programming practice (buffer overflows!)
Programming-language threats
Compiler language inadequacies
Compiler design and implementation flaws, buffer overflows
Hardware threats
Flaws in hardware design and implementation
Undesirable internal hardware state alterations
Improper hardware maintenance
Inside-environmental threats
Internal power disturbances
Self-generated or other internal interference

Discussion Topics: Threats

- -

Are single-point failures and single-flaw exploitations intrinsically different from multiple failures and multiple-flaw exploitations, with respect to the threats? With respect to risks? What about with respect to system architecture (preliminary guess)? (Architecture issues are of course considered later in the course.)

What inherent differences might there be between intentional threats, accidents, and environmental causes? With respect to threats, risks, and architectures...

Reading for the Next Class Period

- -

Read Chapter 3 of the arl-one report on requirements for survivability:
http://www.csl.sri.com/neumann/arl-one.html.

Requirements for Survivability

ENPM 808s
Information Systems Survivability:
4. Requirements for Survivability

- -

Requirements for systems survivability and their relationship with subtended requirements for security, reliability, performance, et al.

Security Requirements Must Consider

- -

Protection against realistic security threats

Protection against malice and accidents

Protection against insiders and outsiders

System security and information security, tamperproofing, hindering reverse-engineering

Integrity, confidentiality, authentication, authorization, monitoring, accountability, prevention and detection of misuse including Trojan horses and denial-of-service attacks

Note: Existing security criteria cover only a small fraction of a loaf.

Reliability Requirements Must Consider

- -

Protection against realistic reliability threats

System reliability and availability

Process reliability and availability

Information reliability and availability

Different degrees of fault tolerance and guaranteed availability, including fault detection, fault avoidance, dynamic fault elimination, fault tolerance (e.g., fail-safe, fail-soft, fail-fast, fail-secure), fault recovery (forward or backward), ...

High-Performance Computing

- -

Performance in the past has dominated other requirements, often to the exclusion of survivability-related issues. Unfortunately, its tradeoffs with other requirements are often severe, and have led to those other requirements being seriously neglected.

John Hennessy (a long-time HPCC leader) recently noted that performance needs to share the spotlight with "availability, maintainability, and other qualities." (*IEEE Computer 32,* 8, pp. 27–33, August 1999).

Other Survivability-Related Requirements

- -

Functional correctness where necessary

Interoperability of (sub)systems

Reconfigurability under duress

Ease of use

Predictability of behavior

Many of these requirements interact with one another, often interfering or creating undesirable tradeoffs. All of the requirements and their interactions need to be considered in advance.

Other Issues

- -

Assurance of required performance

System and information access

Acceptable alternatives in crisis, with static and dynamic priorities

Prevention of malicious and accidental denials of service; failures due to faults that exceed the coverage of fault tolerance; ensuring adequate performance despite intentional acts, unintentional acts, system malfunctions, and acts of God

Interrelationships Among Requirements

- -

Survivability depends on security, reliability, performance, availability, etc. It requires a total-system view over all functional layers.

Availability depends on security, reliability, performance, ...

Security, reliability, and performance can depend on each other.

Some commonality would be beneficial in implementation, but it is not common!


```
                     Survivability     [An overarching
                          /|\        requirement: a
                         / | \     collection of
                        /  |  \   emergent properties]
                       /   |   \
                      /    |    \
                     /     |     \
                    /      |      \
                   /       |       \
                  /        |        \
                 /         |         \
Major           /          |          \
sub-       Security  Reliability  Performance
requirements  /|\         /|\         /|\
             / | \       / | \       / | \
            /  |  \     /  |  \     /  |  \
           /   |   \   /   |   \   /   |   \
       Inte- Conf- Avail  FT Fail  RT  NRT Avail
      grity id'ity   *    |\ modes /\  /|\   *
        /|   |\      |\   | \   /|   \/     /|\
       / |   | \     | \  |  \ / | Prior-  / | \
      /  #   |  \    |  \     #    ities  /  |  \
    MLI No MLS Dis- MLA  \    No         /  [More detailed
    / change | cret- |    \  change     /   requirements]
   /    /|   | ion-  |     \           /
  /    / |   | ary   |      * Unified *   FT=fault tolerance
 /    /  |   |   |   |      availability  RT=real-time
X  Sys Data  X       X      requirements  NRT=non-real-time
    /|   |\                [X = Shared components of MLX!!]
   / |   | \            [* = Reconvergence of availability]
  /  |   |  \         [# = Reconvergence of data integrity]
```


Survivability Attributes at Different Logical Layers (1)

Funct layer	Security concepts	Reliability concepts	Performance concepts
User admin	Human integrity, educ/training, ...user identity	Human reliability educ/training, human interfaces	Responsiveness, ease of use, educ/training
Apps	Application integrity and confidentiality	Funct'l correctness, redundancy, robustness	App availability, real-time perf., funct'l timeliness
Middle ware	App integrity + confidentiality in DCE, Webware, DB access control	Funct'l correctness, redundancy, backup, recovery, robustness	Funct'l timeliness DB of Web, remote DBs, file servers
Net	Net integrity, confidentiality, availability, nontamperability, peer authentic'n, esp. wireless	Net integrity, error correction, fault tolerance in transmission+ routing, espec. wireless roving	Netware throughput, service nondenial, alternative routing, other infrastructure factors, especially bandwidth

Survivability Attributes at Different Logical Layers (2)

Funct layer	Security concepts	Reliability concepts	Performance concepts
OS	OS integrity, data confid'ity, service nondenial, OS nontamperability, OS development and maintenance, OS user authentication	OS integrity, fault tolerance, sound asynchrony, archiving/backup OS development + maintenance	OS integrity, service nondenial, deadlock avoidance, perf. optimization, OS development and maintenance
HW	Access protection domains, HW nontamperability, config. control, protection against interference, HW development	Fault tolerance, instruct'n retry, error-corr codes, HW correctness, protection vs interference, HW development	Proc./memory speed, comm. bandwidths, contention control, HW configuration, protection against all interference, HW development

Discussion Topic: System Safety vs Survivability

- -

System safety is in some ways very similar to system survivability.

It is an emergent property of the system as a whole, and is not generally meaningful in terms of components.

It depends on security, reliability, performance, and other system attributes, and indeed may depend on system survivability!

Here is a topic that is worthy of some discussion: Contrast system safety with system survivability, with respect to risks, threats, requirements, and the implications on design and implementation.

A good reference on system safety is Nancy Leveson's book, *Safeware: System Safety and Computers,* **Addison-Wesley, 1995.**

Discussion Topic: Role of Requirements

- -

What are the potential benefits of having requirements well-defined in advance?

What might we do to ensure that we have the proper requirements from the outset?

Reading for the Next Class Period

- -

Chapter 4 of arl-one (systemic inadequacies):
http://www.csl.sri.com/neumann/arl-one.html.
Also, reflect on Chapters 1 through 4 of *Computer-Related Risks* **to see where you think the most important inadequacies lie. Pick some examples of design flaws and implementation bugs that you consider particularly survivability relevant, and be prepared to discuss them. You may find the Illustrative Risks summary list useful, with its descriptors.**

Systemic Inadequacies and Other Deficiencies

ENPM 808s
Information Systems Survivability:
5. Systemic Inadequacies and Other Deficiencies

- -

We consider a wide variety of deficiencies, particularly in the light of some of the more illustrative Illustrative Risks cases.

Deficiencies in Existing Systems

- -

Requirements are often wrong, designs are flawed, code is buggy, maintenance is flaky.

Commercial products are often inadequate. Certain components are missing that are necessary for survivable systems and networks.

Further research and prototype development are needed.

Systems tend to be overly dependent on human frailty.

There are many other limitations as well.

COTS, Firewalls, etc.

- -

Existing commercial products are seriously deficient, with respect to functionality, reliability, security, and survivability of operating systems, servers, backup systems, network protocols and their implementations, authentication, cryptographic protocols and their implementations, real-time monitoring and analysis; the same is true for ease of use, interoperability, compatibility, etc. These deficiencies are not well understood.

Firewalls and other boundary protection techniques tend to be permeable. E-mail, PostScript, html, ActiveX pass through.

Software Development

- -

Software development practice is abysmal. The entire development cycle is riddled with problems.

Standards, evaluation criteria, and guidelines fall far short of what is needed.

System concepts are often not defined in advance.

Requirements tend to be poorly defined, incomplete.

Designs are typically incompletely specified and frequently inherently flawed.

Implementations are astoundingly often buggy and not consistent with their designs.

Debugging and testing are typically sloppy, and inherently incomplete. ("Testing demonstrates only the presence of bugs, never the absence of bugs.") Quality assurance is shunned.

Good existing research and development is very slow to appear in commercial systems.

Management is often oblivious to the problems of system development.

Routine maintenance and long-term system evolution are seldom planned in advance.

Risk management is seldom adequate.

Procurement practice is full of pitfalls.

Above all, people are inherently fallible.

The Patriot air-defense system. It was assumed in the design that the launch platforms would be moved and the computers rebooted once a day. Therefore, serious clock drifts were consistent with the requirements.

Pacemaker deaths due to electromagnetic interference

The Year-2000 Problem!

Whose shortsightedness do we blame for these?

Examples of Design Vulnerabilities

- -

The ARPAnet collapse of 1980 (software design weakness, lack of redundancy in hardware, dropped bits in memory)

The C-Compiler Trojan Horse, Ken Thompson, 1984 (compiler is part of the trustworthiness perimeter)

The ARPAnet collapse of 1980

In the 1970s, the ARPAnet was a network linking primarily research computers, mostly within the United States, under the auspices of the Advanced Research Projects Agency (ARPA) of the Department of Defense (DoD). (It was the precursor of the Internet, which now links together many different computer networks worldwide.) On October 27, 1980, the ARPAnet experienced an unprecedented outage of approximately 4 hours, after years of almost flawless operation. This dramatic event is discussed in detail by Eric Rosen (Rosen 1981), and is summarized briefly here.

The collapse of the network resulted from an unforeseen interaction among three different problems: (1) a hardware failure resulted in bits being dropped in memory; (2) a redundant single-error-detecting code was used for transmission, but not for storage; and (3) the garbage-collection algorithm for removing old messages was not resistant to the simultaneous existence of one message with several different time stamps. This particular combination of circumstances had not arisen previously. In normal operation, each net node broadcasts a status message to each of its neighbors once per minute; 1 minute later, that message is then rebroadcast to the iterated neighbors, and so on. In the absence of bogus status messages, the garbage-collection algorithm is relatively sound. It keeps only the most recent of the status messages received from any given node, where *recency* is defined as the larger of two close-together 6-bit time stamps, modulo 64. Thus, for example, a node could delete any message that it had already received via a shorter path, or a message that it had originally sent that was routed back to it. For simplicity, 32 was considered a permissible difference, with the numerically larger time stamp being arbitrarily deemed the more recent in that case. In the situation that caused the collapse, the correct version of the time stamp was 44 [101100 in binary], whereas the bit-dropped versions had time stamps 40 [101000] and 8 [001000]. The garbage-collection algorithm noted that 44 was more recent than 40, which in turn was more recent than 8, which in turn was more recent than 44 (modulo 64). Thus, all three versions of that status message had to be kept.

From then on, the normal generation and forwarding of status messages from the particular node were such that *all of those messages* and their successors with newer time stamps had to be kept, thereby saturating the memory of each node. In effect, this was a naturally propagating, globally contaminating effect. Ironically, the status messages had the highest priority, and thus defeated all efforts to maintain the network nodes remotely. Every node had to be shut down manually. Only after each site administrator reported back that the local nodes were down could the network be reconstituted; otherwise, the contaminating propagation would have begun anew.

This case is considered further in Section 4.1 of *Computer-Related Risks*, and further explanation of the use of parity checks for detecting any arbitrary single bit in error is deferred until Section 7.7 of the book.

The C-Compiler Trojan Horse

Ken Thompson's 1983 Turing Award lecture published in 1984 included the now-classical Trojan horse involved a modification to the object code of the C compiler such that, when the login program was next compiled, a trapdoor would be placed in the login object code.

No changes were made either to the source code of the compiler or to the source code of the login routine. Furthermore, the Trojan horse was persistent, in that it would survive future recompilations. Thus, it might be called a *stealth* Trojan horse, because there were almost no visible signs of its existence, even after the login trapdoor had been enabled. This case was, in fact, perpetrated by the developers of Unix to demonstrate its feasibility and the potential power of a compiler Trojan horse, and not as a malicious attack. But it is an extremely important case because of what it demonstrated.

Master-password trapdoor, with a buffer overflow,

The AT&T long-distance collapse of 1990

Master-password trapdoor

Master-password trapdoor, with a buffer overflow, Young and McHugh, p. 115-116 of *Computer Related Risks*

Young and McHugh describe an astounding flaw in the implementation of a password-checking algorithm that permitted bogus overly long master passwords to break into every system that used that login program, irrespective of what the real passwords actually were and despite the fact that the passwords were stored in an encrypted form. This rather subtle flaw and its exploitation are depicted in the figure. It involved the absence of strong typing and bounds checking on the field in the data structure used to store the user-provided password (field b in the figure). As a result, typing an overly long bogus password overwrote the next field (field c in the figure), which supposedly contained the encrypted form of the expected password. By choosing the second half of the overly long sequence to be the encrypted form of the first half, the attacker was able to induce the system to believe that the bogus password was correct.

```
Internal Data Structure
= = = = = = = = = = = = = = =
a. User login name
b. User-typed PW
c. Stored encrypted PW
d. Encrypted form of typed PW

The Password-Checking Algorithm
= = = = = = = = = = = = = = = =
1. USER TYPES login name --> a
2. Stored encrypted PW --> c
3. USER TYPES PW --> b
4. Encrypt user typed PW --> d
5. Compare c and d
```



```
Step 1.     3.       2.          4.
  ---------------------------------------
  Store | Store | Stored    | Encrypted
  login | typed | encrypted | typed
  name  | PW    | PW        | PW
  = = = = = = = = = = = = = = = = = = = =
  |   a   |   b   |     c     |    d    |
  = = = = = = = = = = = = = = = = = = = =
        Step 5 compares c and d.
```



```
   The Master-Password Attack
   = = = = = = = = = = = = = = = = =
   EMEMY CHOOSES a string --- "opqrst".
   ENEMY ENCRYPTS string, gets "uvwxyz"
1. ENEMY TYPES any legitimate user name,
     which is entered into field a.
2. The stored encrypted PW goes into c.
3. ENEMY types PW "opqrstuvwxyz" ---
     which is entered into b (and c!).
4. Field b is encrypted and stored in d.
5. Surprise!  Fields c and d match.
```



```
     The Perfect Match!
   ---------------------------
   | a |   b  | c    | d    |
   ---------------------------
1.|Name|      |      |      |
2.|Name|      |ghijkl|      |
3.|Name|opqrst|uvwxyz|      |
4.|Name|opqrst|uvwxyz|uvwxyz|
5.             uvwxyz=uvwxyz
```

Master-password flaw and attack (PW = password)

AT&T long-distance collapse of 1990

In mid-December 1989, AT&T installed new software in 114 electronic switching systems (Number 4 ESS), intending to reduce the overhead required in signaling between switches by eliminating a signal indicating that a node was ready to resume receiving traffic; instead, the other nodes were expected to recognize implicitly the readiness of the previously failed node, based on its resumption of activity. Unfortunately, there was an undetected latent flaw in the recovery-recognition software in every one of those switches.

On January 15, 1990, one of the switches experienced abnormal behavior; it signaled that it could not accept further traffic, went through its recovery cycle, and then resumed sending traffic. A second switch accepted the message from the first switch and attempted to reset itself. However, a second message arrived from the first switch that could not be processed properly, because of the flaw in the software. The second switch shut itself down, recovered, and resumed sending traffic. That resulted in the same problem propagating to the neighboring switches, and then iteratively and repeatedly to all 114 switches. The hitherto undetected problem manifested itself in subsequent simulations whenever a second message arrived within too short a time. AT&T finally was able to diagnose the problem and to eliminate it by reducing the messaging load of the network, after a 9-hour nationwide blockade. With the reduced load, the erratic behavior effectively went away by itself, although the software still had to be patched correctly to prevent a recurrence. Reportedly, approximately 5 million calls were blocked.

The ultimate cause of the problem was traced to a C program that contained a `break` statement within an `if` clause nested within a `switch` clause. This problem can be called a programming error, or a deficiency of the C language and its compiler, depending on your taste, in that the intervening `if` clause was in violation of expected programming practice.

Some Security Deficiencies

- -

Weak operating systems, network protocols, compilers, administrative tools

Poor authentication, poor authorization, lack of layered defenses, few denial-of-service defenses, poor monitoring and real-time analysis,

Poor crypto and crypto embeddings

Poor software-engineering practice, terrible human-interface design

Poor operational/admin practice

Human Screw-ups

- -

Shuttle Challenger loss. The decision was made to launch in cold weather despite known risks of O-ring failure. Environmental problem. Specification lacked foresight?

Patriot inaccuracies. Rebooting the platform daily would have overcome the excessive clock drift. Requirement failure? Software bug? All of the above...

Aegis system's role in U.S.S. Vincennes' shootdown of an Iranian Airbus (human interface problems, limited information on screen). Requirement specification failure? User interface design? Operator panic? All of the above.

More Human Screw-ups

- -

Numerous airplane disasters blamed on pilot error and air-traffic controller error KAL 007. NW 255. British Midland. Airbus A320s.

Numerous other transportation accidents: railroad crashes, Exxon Valdez on autopilot.

Blame is Often Widely Distributable

- -

Therac 25: omitted hardware interlock, reliance on inadequate software, operational practice, developer intransigence, etc.

Vincennes' Aegis: archaic hardware and software, bad human interface, use of system outside of its intended operation, etc.

Y2K as a long-term pervasive problem: incredible short-sightedness, even now.

Computer security vulnerabilities: endemic lack of understanding, false economies, lack of management and government pressures, etc.

Discussion Topics: Inadequacies

- -

What are the most serious inadequacies in systems today?

On what kind of systems might you be willing to trust your life? Are you confident *you* could do better than the norm?

What difficulties arise in trying to allocate blame?

What good and what harm might arise from attempting to allocate blame?

Apply this discussion to the Year-2000 problem!

What have you learned thus far, and what are your preliminary conclusions?

Reading for the Next Two Class Periods

Read Chapter 5 of arl-one on approaches for overcoming such deficiencies: http://www.csl.sri.com/neumann/arl-one.html

Approaches for Overcoming Deficiencies 1

ENPM 808s
Information Systems Survivability:
6. Approaches for Overcoming These Deficiencies

What exists (not enough), and what is needed – architecture, components, configuration management, infrastructures, protocols, standards and criteria, above all, a coherent collection of defensive measures that can be seamlessly integrated

NOTE: Lectures 6 and 7 are paired. The exact split as to what goes in which lecture is not important.

Typical Defensive Measures

Approach	Relability	Security	Performance
Prevent	Architectural robustness: Redundancy, Error correct'n Fault tolerance in HW and SW Synchronization Generalized dependence	Architectural robustness: Isolation, Authentication Authorization Differential access ctls Generalized dependence	Architectural robustness: Spare capacity, Reconfigurability Multiprocessing and highly distributed systems
Detect	Redundancy: Error detection in HW and SW	Integrity checking; Anomaly+misuse detection	Performance monitoring
React	Forward/backw'd recovery in HW and SW	Security-preserving reconfiguration	Reconfiguration
Iterate	Redesign, reimplement: Fault removal or tolerance	Redesign, reimplement: Exploratory code patches	Redesign, reimplement: Experimental tradeoffs

Protocols

The existing Internet protocols (TCP, IP, ftp, telnet, udp, smtp etc.) must be extended, revised, augmented, and/or replaced to foster greater robustness.

IPSEC and IP Version 6 are seriously incomplete. They provide end-to-end encryption, but are inadequate with respect to user authentication, integrity, prevention of denials of service, key management, and other issues that are fundamental to achieving meaningfully robust systems and networks. They treat reliability and survivability issues only minimally.

Evaluation Criteria

U.S. evaluation criteria for secure systems (TCSEC, Orange Book, Rainbow Series) are fragmentary, seriously deficient for networks and distributed systems. Incomplete. Very narrow view of security. Overly constraining. Very weak on authentication and integrity, insider misuse, Trojan horses. Tolerates fixed passwords! Static. Poorly matches functionality and assurance. Very outdated. MLS oriented. Software engineering and reliability almost nonexistent. (See PGN, Rainbows and Arrows: How the Security Criteria Address Computer Misuse, NatlCompSecConf 1990.)

Evaluation Criteria (more)

The Common Criteria are emerging for security (http://csrc.nist.gov/cc), following the Canadian (CTCPEC) and European (ITSEC) criteria. However, these are merely frameworks, with roll-your-own specifics against which to be evaluated. Functional and assurance requirements are separated (good).

There are as yet no adequate criteria for survivable systems and networks. Interoperability, configurability, and many other requirements are absent.

Standards and criteria are important, but beware of bad ones and good ones misapplied.

Types of Computer Misuse (review)

External misuse (EX)
1. Visual spying
2. Misrepresentation
3. Physical scavenging

Hardware misuse (HW)

4. Logical scavenging
5. Eavesdropping
6. Interference
7. Physical attack
8. Physical removal

Masquerading (MQ)

9. Impersonation
10. Piggybacking attacks
11. Playback and spoofing
12. Network weaving

Pest programs (PP)

13. Trojan-horse attacks)
14. Logic bombs
15. Malevolent worms
16. Virus attacks

Bypassing authentication/authoriz'n (BY)

17. Trapdoor attacks
18. Authorization attacks

Active misuse of authority (AM)

19. Creation, modification, use, denials
20. Incremental attacks
21. Denials of service

Passive misuse of authority (PM)

22. Browsing randomly or searching
23. Inference and aggregation
24. Covert channel exploitation
25. **Misuse through inaction** (IM)
26. **Use as an indirect aid** (IN)

Orange Book\ Misuse Criterion \Technique	EX 1-3	HW 4-8	MQ 9-12	PP,BY 13-18	AM 19-21	PM 22	PM 23	PM 24	IM 25	I 2
TCSEC Security Policy:										
Discretionary access control				(*)	*	*				
Object reuse				17b,c						
Labels & label integrity				*	*	*	*	*		
Exportation (3 criteria)		(4)		*	*	*	*	*		
Labeling human output	(3)			*	*	*	*	*		
Mandatory access control				*	*	*	*	*		
Subject sensitivity labels				*	*	*	*	*		
Device labels				*	*	*	*	*		
TCSEC Accountability:										
Identification/authentication			(9)	*	*	*	(*)	(*)		
Audit			(*)	*	*	*	*	*	(*)	(
Trusted path			*	*						
TCSEC Assurance:										
System architecture			(*)	*	*	*	*	*		
System integrity				*						
Security testing			(*)	(*)						
Design spec/verification				*	*	*	(*)	*		
Covert channel analysis						(*)	(*)	*		
Trusted facility mgnt	(3)			(*)	*	*	(*)			
Configuration management			(*)	*						
Trusted recovery			(*)	*						
Trusted distribution				*						
TCSEC Documentation:										
Security features user's guide				*	*	*	*	*		
Trusted facility manual				*	*	*	*	*		
Test documentation				*	*	*	*	*		
Design documentation				*	*	*	*	*		

Future Needs

Detailed functional requirements encompassing realistically complete sets of survivability relevant requirements (security, reliability, fault tolerance, etc.) that could be used for a wide range of procurements, presumably parametrically rather than generically

Detailed guidance on how to handle the interactions among the various constituent subtended requirements for survivability

Design frameworks and alternative families of logical architectures and architectural structures that facilitate the composition of survivable system and networks out of subsystems

Guidance on good system engineering and software engineering practice, and far-reaching approaches toward system development

Future Needs, continued

Guidance on good procurement practice

Guidance on achieving interconnectivity and reusability

Guidance on hardware-software tradeoffs, particularly with respect to crypto implementations. (There is a great need for hardware-based authentication as in Fortezza, but hardware-based crypto is by no means a panacea.)

Future Needs, continued

Identification of interoperable development environments and constructive tools. However, it must be recognized that tools are not a panacea; their use may in fact be counterproductive.

Identification of fundamental gaps in the existing standards, and identification of which standards are or are not truly compatible with which others

Identification of fundamental components that are absent or inadequate in the existing products, and guidance on what is actually commercially available rather than merely a hopefully emerging product.

Missing Components

Secure and robust boundary controllers (not just firewalls based on weak operating systems that pass executable content), network controllers, and server systems. To some extent, we can do without really secure and robust client systems (workstations, laptops, palmtops) if we have strong servers.

Cryptographic co-processors and distributed authentication servers using nontrivial passwords (e.g., trustworthy certificates, smart cards, cryptographically generated tokens)

Bilateral “trusted path” mechanisms (must be trustworthy)

Multilevel-secure servers when MLS is a necessity

See subsequent lectures on architecture.

Defenses Against Compromise

Exogirding vs. compromise from outside: Firewalls, authentication, encryption, electromagnetic shielding

Endogirding vs. compromise from within: Compile-time and run-time checks, fault tolerance, cryptographic authentication and integrity checks, monitoring and real-time misuse/anomaly detection

Undergirding vs. compromise from below: security kernels and trusted computing bases, trustworthy operating systems, special-purpose hardware and co-processors, integrity checks

Multilevel Security (MLS)

Information may not flow from one entity at a particular security level to another entity at a lower security level.

An entity could be a process, a file, a server, an entire system or network, or any other computer resource.

As a mandatory policy, MLS controls flow to within levels and compartments. Ideally, if properly implemented in a security kernel or trusted computing base, it cannot be subverted by users or any higher-layer entities.

Problems with MLS

Mandatory security policies are not well supported in practical commercial systems, difficult to use, difficult to configure properly, and operationally require many exception mechanisms that must be trusted. High-watermark policies tend to drive information to higher classification levels.

The Orange Book MLS criteria are incomplete, and MLS is only a small part of the problem.

Multilevel Integrity (MLI)

An entity at a particular integrity level may not depend for its correctness (the Parnas *uses* relation) on another entity at a lower integrity level.

An entity could be a process, a file, a server, an entire system or network, or any other computer resource.

As a mandatory policy, it restricts the damaging effects of Trojan horses to within levels and compartments. However, it seems awkward to use without many trusted exceptions, because of vast dependence on untrusted editors, compilers, other utilities. Web-of-trust arguments tend to lead to low-watermark integrity.

Multilevel Availability (conceptual)

An entity at a particular availability level may not depend on another entity at a lower availability level.

An entity could be a process, a file, a server, an entire system or network, or other computer resource.

As with MLS and MLI, MLA would have availability levels associated with each entity.

Multilevel Survivability (conceptual)

An entity at a particular survivability level may not depend on another entity at a lower survivability level.

An entity could be a process, a file, a server, an entire system or network, or any other computer resource.

Combined with generalized dependence, this could be a useful architectural organizing concept, although it is probably too restrictive for entire systems and networks, even if combined with generalized dependence.

A Few Comments

An Orange Book reference monitor is supposed to be nonbypassable, tamperproof, and small enough to be analyzable. Typical kernels and TCBs are not.

The Java Security Manager mobile code is not consistent with those three needs, and hinders mobile-code survivability.

Windows NT is reportedly 48 million lines of source code (with lots of memory required to load it and run it), plus test code (7.5M lines)

The metaphor that you cannot make a silk-purse out of a sow's ear seems quite apt, although there are a few examples in fault tolerance.

Approaches for Overcoming Deficiencies 2

ENPM 808s
Information Systems Survivability:
7. Overcoming These Deficiencies 2

Role of security and fault tolerance, software engineering, good development practice, principles, good experience, selective use of formal methods

This lecture is in memory of John Gannon of the Maryland Computer Science Department.

NOTE: Proposals for your final projects are due by e-mail the day after the NEXT class (in ASCII, PostScript, pdf, html, or other format that I can read easily on a Unix-based systems).

Reminder of Serious Obstacles

Existing system development practice is generally terrible.

The tendency toward COTS products is a self-defeating prophecy unless dramatic improvements occur.

Development Tools Can Help

- -

Record design decisions
Enforce desired design structure
Check syntax of specifications
Record/control development status
Provide programming environment (compilers, optimizers, debuggers, test data generators, editors, symbolic executors, simulators, etc.)
Prove design fulfills requirements
Prove code consistent with specs
Perform refining transformations
Generate and configure systems

Good software engineering practice (e.g.,)

- -

Modular design
Functional abstraction
Well-specified functionality
Reusable and interoperable interfaces
Information hiding to mask implementation detail
Constructive use of analysis techniques and tools, selective use of formal methods
Higher-level programming languages less susceptible to characteristic errors (missing bounds checks, mismatched types, mismatched pointers, off-by-one errors, missing exception conditions)
Good documentation


```
   Desired Characteristics
of Survivable Systems and Networks
----------------------------------
secure            reliable
available         responsive
flexible          monitorable
evolvable         reusable
portable          convenient
understandable    simple/realistic
uniform           forgiving
efficient         cost-effective
compatible        usefully documented
interoperable     rapidly prototypeable
```


Schopenhauer:
Experience and Principles

- -

.. our heads are full of general ideas that we are now trying to turn to some use, but that we hardly ever apply rightly. This is the result of acting in direct opposition to the natural development of the mind by obtaining general ideas first, and particular observations last; it is putting the cart before the horse. ...

The mistaken views ... that spring from a false application of general ideas have afterwards to be corrected by long years of experience; and it is seldom that they are wholly corrected. That is why so few men of learning are possessed of common sense, such as is often to be met within people who have had no instruction at all.

Arthur Schopenhauer, *Parerga and Paralipomena,* **1851.**

Einstein: Thinking and Judgement

- -

The development of general ability for independent thinking and judgment should always be placed foremost, not the acquisition of special knowledge. If a person masters the fundamentals of his subject and has learned to think and work independently, he will surely find his way and besides will better be able to adapt himself to progress and changes than the person whose training principally consists in the acquiring of detailed knowledge.

Albert Einstein, *Out of My Later Years,* **1950.**

Security Design Principles
(after Schroeder-Saltzer 1972)

0. Methodological basis
1. Simplicity
2. Fail-safe defaults
3. Complete mediation
4. Open design
5. Separation of privilege
6. Principle of least privilege
7. Principle of least common mechanism
8. Psychological acceptability
9. Sufficiently high work factor
10. Noncompromisible auditing
11. Principle of least surprise
(especially in interfaces)

0. Unifying Methodological View

Development Stages	Analysis stages: Functional	Performance
Requirements (functional+ performance)	Consistency, adequacy, realizability	Consistency, adequacy, realizability
Design: Hierarchy, Specifications (func+perf)	Design proofs (spec-req consistency) Design reviews	Design feasibility (spec-req consistency)
Implemen-tation: Code, linkage	Code proofs (code-spec consistency, portability checks), Tiger teams	Performance test and measurement Formal analysis as appropriate

Development Stages	Analysis stages: Functional	Performance
Maintenance	Incremental closure after change	Check for adequacy of performance of revised system
Documentation	Consistency of documentation with design, code	Consistency of documentation with design,code

1. SIMPLICITY

Einstein:
Everything should be made as
simple as possible, but no simpler.

* Conceptual simplicity,
  economy of mechanism:
  Keep it simple, Keep it small.
  (But complex systems are not simple.)
* Comprehensibility of design
* Controllability of implementation
* Comprehensibility of validation
* Constrained code development
* In-house expertise

All of these suggest the use of high-level
design languages and programming languages.
WE NEED VIRTUALIZED SIMPLICITY.

2. ACCESS-CONTROL DEFAULT

* Protection scheme permits access unless
  denied (typically uncontrollably dangerous)
* Protection permits no access unless
  specifically authorized (typically too
  restrictive and causes many error messages)
* Multilevel security with preassigned levels
  (conceptually simple, but full of pitfalls)

3. COMPLETE MEDIATION

* EVERY access must be checked
* Must be system-wide
* ''Foolproof'' identification
* Audit changes in authority
* Reference Monitors

The Reference Monitor Concept
(Anderson, 1972)

Reference monitor mediates security
on every access by subject to object.
May use hardware and software.

Small enough be analyzed thoroughly
(formal verification?)

Protected from tampering,
nonbypassable.

4. OPEN DESIGN

* Design should not be a secret.

* Protection mechanisms separate from protection keys

  - Pins and tumblers vs. combinations
  - Security design vs. passwords

* Provision for open review of protection mechanism

* Offer bounty for penetrations

5. SEPARATION OF DUTIES

* Enforce distinct roles, Firewalls of protection

* Require multiple authorizations, not all in one (avoid superuser)

6. LEAST PRIVILEGE

* Require need-to-know

* Allocate just what is needed

7. LEAST COMMON MECHANISM

* Minimize amount of mechanism common to more than one user.

* Minimize pervasiveness of shared variables.

8. PSYCHOLOGICAL ACCEPTABILITY

* Easy to use and understand, no incentive to circumvent.

9. WORK FACTOR

* Related to degree of difficulty of cracking

* Increased possibility of exposure

10. NONCOMPROMISIBLE AUDITING

* Need to record all interesting events, particularly potential compromises (survivability/security/reliability)

* Thresholds depending on context

* Must be tamperproof: nonbypassable, nonalterable

11. PRINCIPLE OF LEAST SURPRISE

* Relevant especially to systems in the large

* Vital in human interfaces (user interface coherence), and also in accessible internal program interfaces

Implementation Principles

Superb design can be compromised by shoddy implementation.
Good implementation is vital.
Implementation structure should relate to design.
Use of appropriate higher-level languages
Use of appropriate hardware
Defer binding until essential.
Defer optimization until needed.
Anticipate later problems.

```
Good Programming Language Features
----------------------------------
Compatibility with methodology
Ease of programming, understanding
Efficient code compilable
Support for separate compilation
Encapsulation of data types
Strong typing and type safety
Symmetry (built-in, extensions)
Dynamically declarable attributes
Dynamic creation and deletion
Ease of expressing concurrency
Sound synchronization primitives
Clean handling of exceptions
Initialization and finalization
Controlled argument passing
No aliases
```



```
Riskful Programming Language Problems
-------------------------------------
Nonisolated dynamic memory management
Multitasking (synchronization, process
  isolation)
Dependence on untrustworthy runtime
  libraries
Do-it-yourself exception handling
GoTos and nonnested calling semantics
Unchecked arguments, bounds, arrays,
  buffer overflows
Lack of strong typing
Unchecked escapes into machine code
Leaky garbage collection
Undetectable Trojan horses
```


Roles of Formal Methods in Survivability

Formal methods can play an important role in the attainment of systems and networks that must achieve generalized survivability, in specification, design, and execution. Great improvements in system behavior can be realized when the requirements (for survivability, security, reliability, etc.) have a formal basis. Similarly, there are enormous benefits whenever design specifications have a formal basis, especially if they are derived from well specified requirements rather than the common practice of being established after the fact to represent an ad hoc assembly of already developed software (sometimes referred to as putting the cart before the horse). Formal design verification then involves formal demonstrations that the specifications are consistent with their requirements, providing no less than required, and to the extent that the absence of Trojan horses can be demonstrated, nothing unexpected that might be harmful. Verification of designs is difficult for systems that were not designed to be readily analyzed, but can nevertheless be valuable in legacy systems (as in analyses of the risks associated with the Year-2000 problem). Finally, although it is less commonly practiced in software, formal code verification can demonstrate that a given implementation is consistent with its specifications. Formal hardware verification is being used increasingly, and demonstrates the potential effectiveness of formal methods where there are considerable risks (financial or otherwise) of improper design and implementation.

Various formal methods can be valuable in specifying and analyzing requirements, designs, and implementations. Of particular importance in connection with survivability are techniques that can provide formal relationships between different layers of abstraction, with respect to requirements and specifications alike. The use of formal methods is recommended in particularly critical applications, and can help move the current highly unpredictable ad hoc development process into a much more predictable formal development process. In the long run, use of such techniques can dramatically decrease the risks of system failure. Contrary to popular myth, judicious use of formal methods can also decrease the overall development and operating costs, especially when the costs of aborted developments (e.g, cancellations of IRS, FAA, FBI systems) are considered, along with the costs of overruns, delays in delivery, and subsequent maintenance.

Judicious use of formal methods can have a very high payoff, particularly in requirements, specifications, algorithms, and programs concerned with especially critical functionality, such as concurrency, synchronization, avoidance of deadlocks and race conditions in the small, and perhaps even network stability and survivability in a larger context, derived on the basis of more detailed analyses of components. There is no substitute for using demonstrably sound algorithms.

Of particular importance is the formal analysis of requirements, for example, determining whether a given set of requirements at a particular layer of abstraction is consistent within itself, whether the different sets of requirements at the lower layer are fundamentally incompatible with one another, and whether the requirements at a lower layer are consistent with the requirements at the upper layer. Once such an analysis is done, then it is also beneficial to determine whether system specifications and implementations are consistent with the relevant requirements.

It must be emphasized that the most valuable uses of formal methods are in finding flaws and inconsistencies, not in attempting to prove that something is completely correct. However, formal methods approaches are not absolute guarantees, because problems that can exist outside of their scope of analysis. For example, suppose that a given analysis does not detect any flaws or inconsistencies in a specification or implementation. It is still possible that the requirements are inadequate (for example, the specifications could fail to prevent a problem not covered by the requirements), or that the analysis methods themselves could be flawed. For these reasons, extensive testing of developed systems is also important – albeit inherently limited. Unfortunately, testing is itself inherently incomplete and incapable of discovering many types of problems, for example, stemming from distributed system interactions and concurrency failures, subtle timing problems, unanticipated hardware failures, and environmental effects. Exhaustive testing over all possible scenarios is basically impossible in any complex system.

Past Uses of Formal Methods

Fault tolerance and Byzantine algorithms; SRI's 1970s Software Implemented Fault Tolerance (SIFT) and Provably Secure Operating System (PSOS); security and cryptographic protocols; network protocol development; hardware verification and computer-aided design; compiler correctness; and human safety; but not yet for survivability, or for security and fault tolerance in combination.

The European notion of dependability attempts to encompass various requirements.

Belief logics for cryptographic protocols (e.g., Burrows-Abadi-Needham BAN logic)

Formal methods can also be used in execution: Proof-carrying code, providing evidence that a component has not been tampered. See references to George Necula and others.

Directions for the Future

Establishment of survivable heterogeneous open-system and network architectures, designed for the full set of requirements, with robust information infrastructure, exploiting robust mobile code; focus on selected trustworthy critical components; minimizing trustedness in dumb terminals and handheld wireless communicators; real-time detection of anomalies; robust configuration management

Dramatically improved system development practice and procurement practice

Better use of research and development, sensible fault tolerance, security; formal methods for specification, selected proofs of critical properties of critical modules, proof-carrying code, safe reconfiguration, ...

Establishment of explicit requirements and understanding of the interdependencies

Better system/software engineering practice: disciplined use of structural organizing principles for survivable systems, such as abstraction, composition, encapsulation, reusability, separation, least privilege; domains; baseline survivable architectures; enriched IPSec/IPv6 for realistic survivability requirements; adherence to the Generally Accepted System Security Principles (GASSP) (http://web.mit.edu/security/www.gassp1.html)

Practical development and use of real-time analysis and response subsystems coupled with dynamic configuration control

Find ways to develop the missing pieces of the computer-communication infrastructure, and ruggedize the best R&D available. We need sound operating systems for boundary controllers, crypto, trustworthy servers, pervasive authentication, and architectures that localize critical trustworthiness needs.

Provide better architectural and implementation guidance to avoid characteristic weaknesses.

Improve the evaluation criteria framework (Common Criteria!) and establish specific requirements for evaluation of survivability

General References

- -

Computer-Related Risks and the arl-one report provide an enormous number of references. For example:

Software engineering: Many references to works by Dijkstra beginning in 1969s, Parnas in 1970s, FPBrooks, ...

Robust systems: Multics (1965 and on, http://www.multicians.org), T.H.E. system (Dijkstra 1968), domains of protection (Schroede 1972), confined execution (Lampson 1973), PSOS (1975, 1977, 1980), Rushby's isolation kernels (Rushby 1984).

Early efforts linking security and reliability: Lampson 1974, Dobson and Randell 1986, Randell and Dobson 1986.

Reading for the Class Periods on Architectures

- -

Read Chapter 7 of arl-one:
http://www.csl.sri.com/neumann/arl-one.html

Look at the Feiertag-Neumann PSOS paper for its hierarchical layering and object-oriented architecture:
http://www.csl.sri.com/neumann/psos.pdf

Scan the Proctor-Neumann paper for its architectural content, not for the covert channel analysis:
http://www.csl.sri.com/neumann/ncs92.html

Architectures for Survivability 1

ENPM 808s
Information Systems Survivability:
8. Architectures for Survivability 1

- -

System- and network-oriented approaches; architectures: architectural components, architectural structure, and structural architectures; servers; mobile-code paradigms; composition

System Structure

- -

Highly structured system architectures can have enormous benefits in system specification, development, implementation, maintenance, evolution, networking, and flexibility, and interoperability, for years to come. Useful in protocol suites, scalable multicast protocols, cryptographic key distribution and key management, divide and conquer algorithms, etc. Several historical examples of system designs are worth noting.

System Structure: The T.H.E. System

- -

The T.H.E. (Eindhoven) system, Dijkstra 1968, with a hierarchically locking strategy to prevent interlayer deadly embraces via a linearized synchronization strategy. Deadly embraces still occurred occasionally within a level, but never between levels. [Parnas strict-sense "uses" hierarchy: each layer depends on the correctness of the lower layer.]

Layer 5: The operator
Layer 4: Independent users
Layer 3: Each process has its own virtual keyboard, multiplexed onto a single keyboard, buffering of I/O
Layer 2: Message interpreter controlling operator keyboard
Layer 1: Segment controller for drum interrupt synch
Layer 0: Allocation of processes to processors, respectful of explicit mutual synch

SIFT, A Highly Available System

Software-Implemented Fault Tolerance for commercial flight control (SRI-Bendix-NASA)
Extraordinarily high availability.
7 computers, highly redundant, broadcasting of task results, fault masking by 2/3 voting on critical tasks (3/5 not needed), fault detection and elimination by robust self-diagnosis with deconfigured faulty computers and reallocated tasks
Design & HOL code of paper system formally proven in HDM, redone and extended in EHDM.
System ran for years at NASA Langley.

Proof Refinements for SIFT Paper System
(Melliar-Smith and Schwartz 82)

```
Markov Model Failure Probability 10**(-10)/hr
 /             using HW error-rate analysis
(  I/O Model System SAFE =>
 \    /        "all tasks correct"
Replication  Task replicated;
  Model        values voted upon on
    |          task completion
 Activity    Task actitivies: startup,
  Model        broadcast of values, vote
    |          execute, synchronization
 Operating   SPECIAL specs for OS:
  System       scheduler, voter, dispatcher
    |          buffer manager, etc.
  Pascal     Pascal code for each routine
 Programs
    |
BDX-930 Code [Pascal to BDX not proved.]
```


Multics Hardware (1965!)

Memory mapping for virtual memory
Descriptor for each virtual segment
Access control bits in descriptor
Descriptor cache for performance
Segmentation and paging

Process notion supported in hardware
Process context defined by address space and point of execution, user separation

Rings generalize supervisor/user concept to linearly-ordered domains
Support for argument validation

Original Multics Software (1965!)

Modular: written in PL/I, reentrant, supports process-per-user concept. File system, I/O integrated.

Central access control policy (discretionary)

Unified design philosophy, e.g., dependence on symbolic addressing, dynamic linking of symbolic I/O and file names, segmentation, paging, command standards, conventions, canonicalization, ...

"Hard-core" O/S vs. other user services

O/S reliability/security/survivability based on ring mechanism. Robustness hierarchy.

Multilevel secure retrofit put MLS kernel in Ring 1. 8 levels, 18 categories.

SRI's Provably Secure Operating System

PSOS (a paper system, begun in 1973, with reports in 1975, 1977, and slightly revised in 1980) was apparently the first object-oriented operating system design, in hardware and software. It addressed then-advanced security requirements. Tagged capabilities in hardware, non-kernel multilevel security, hierarchical abstraction. Lower layers (0-12) formally specified. (See Feiertag-Neumann paper on PGN's Web site.)

PSOS Abstraction Hierarchy

Level	PSOS Abstraction or Function
16	user request interpreter *
15	user environments and name spaces *
14	user input-output *
13	procedure records *
12	user processes*, visible input-output*
11	creation and deletion of user objects*
10	directories (*)[c11]
9	extended types (*)[c11]
8	segmentation and windows (*)[c11]
7	paging [8]
6	system processes and input-output [12]
5	primitive input/output [6]
4	arithmetic, other basic operations *
3	clocks [6]
2	interrupts [6]
1	registers (*), addressable memory [7]
0	capabilities *

```
|--------------------------------------------|
|  *  = functions visible at user interface. |
| (*) = partially visible at user interface. |
| [i] = module hidden by level i.            |
| [c11] = creation/deletion hidden by level 11|
|--------------------------------------------|
```

```
     Illustrative PSOS Properties
(Not all are primary security properties)
-----------------------------------------
17. Soundness of user type managers
15. Search path flaw avoidance
12. Process isolation, i-o soundness
11. No lost objects (w/o capabilities)
 9. Generic type safety
 8. Correct segment, no residues
 6. Interrupts properly masked
 4. Correctness!
 0. Nonforgeable, nonbypassable,
    nonalterable (MLS if desired)
```



```
         PSOS Generic Hierarchy
|--------------------------------------|
|Level|    PSOS Abstraction    | Level |
|--------------------------------------|
|  F  | user abstractions      | 14-16 |
|  E  | community abstractions | 10-13 |
|  D  | abstract object manager |   9   |
|  C  | virtual resources      |  6-8  |
|  B  | physical resources     |  1-5  |
|  A  | capabilities           |   0   |
|--------------------------------------|
```



```
       A generic capability:
+-+------+------+----------+-------+-+
|T|Unique|Access|Capability|Address|C|
|A|  ID  |Rights|Data      |Base & |O|
|G|      |      |Type      |Bounds |P|
| |      |      |          |       |Y|
+-+------+------+----------+-------+-+
```

PSOS capability contains Tag, UID. TAG bit identifies it as a capability. UID unique for system lifetime.

PSOS capability strongly types the referenced abstract object. Primitive access rights are known to hardware, others interpreted by datatype type managers.

Address base and bounds define virtual address and extent of data referenced.

Copy bits determine rights of passage of capabilities.

Other capability systems vary...

No tag in descriptor-based systems (capabilities stored separately). Descriptors reusable sooner or later.

Multilevel Security

The mandatory nature of multilevel-security policies is very attractive in principle, but full of difficulties in practice.

Kernels, references monitors, trusted computing bases, untrusted applications

Where multilevel security is vital as an overall system/network property, eschew the Orange Book kernel/TCB paradigm locally in favor of the Proctor-Neumann approach in which end-user systems are single level, and MLS trustworthiness is only in certain servers:
http://www.csl.sri.com/neumann/ncs92.html

This largely avoids the desire for MLS PC-like end-user systems, relying instead on

MLS/MLI servers and highly trustworthy components just where they are essential.

Greater attention is needed for MLS as a global system/network property, while avoiding trying to enforce MLS globally where trustworthiness cannot be assured.

Continue exploration of what is required to achieve a Red Book (Trusted Network Interpretation) for composition.

```
NO ADVERSE FLOW --> (e.g., MLS downward)

  ----------               ----------
  |        |    -------    |        |
  | "High" |--->|Guard|--->| "Low"  |
  | (sec)  |<---|     |<---| (sec)  |
  | (int)  |    |-----|    | (int)  |
  |--------|               |--------|

NO ADVERSE FLOW <-- (e.g., MLI upward)
```


Multiplexed Single-Level MLS Architectures

- -

* Rushby and Randell 1983 Newcastle Connection Distributed Secure System (lots of covert channels)

* Proctor and Neumann 1992 SRI multiple single-level (no user covert channels) http://www.csl.sri.com/neumann/ncs92.html

Can use off-the-shelf single-level client systems with trustworthy servers, providing MLS, MLI, and a sense of MLX.

Note: Naval Research Lab (Kang et al. 97, etc.) approach addresses the ability to read down to lower-level systems (without the corresponding MLI to guard against Trojan horses, etc.) via one-way flow architecture (Davidson 96), NRL Pump (Kang 96), SINTRA, COTS switched client workstations.

Architectures for Trustworthy DBMSs

OS \ DB	Untrusted DB	Trust in DB
Untrusted OS	System high	Dedicated DB: OS encaps'd?
Trusted OS TCB	Restrictive: IPSharp Grohn, Bonyun, Hinke-Schaefer, SeaView MLS only	Extended TCB: SeaView *, ASD-Views**, Honeyw SAT DB, AOG-Gemini DB

* SeaView Exended TCB untrusted for mandatory access ctl, trusted for DAC, DB consistency

** ASD-Views extended TCB encapsulates views, which become trusted

SeaView Design Hierarchy (Lunt et al. 88)

```
Ring, Database Functionality
-------------------------------------
TOTALLY UNTRUSTED:
7. User applications
-------------------------------------
6. User interface, presentation view
   mgr, conventional DBMS functions.
   <MSQL Preprocessor if untrusted>
-------------------------------------
TRUSTED FOR DB DAC, INTEGRITY:
5. DBMS Nucleus Resource Mgr --
   Extended DB DAC, DB consistency,
   MSQL Preprocessor <integ. trusted>
-------------------------------------
TRUSTED SINGLE-LEVEL RELATIONS:
3-4. Discretionary TCB (GEMSOS).
     DAC on relations hidden by DBMS
     in MSQL trusted option
0-2. Mandatory TCB (GEMSOS) used for
     single-level relations
-------------------------------------
MSQL provides virtual multilevel
relations and views.
```


SeaView Simplified Model Hierarchy (Denning et al. 88)

- -

2. TCB MODEL: Database TCB for views, multilevel (virtual) relations, discretionary access, labeling, data consistency, sanitization, reclassification, constraints on transactions, polyinstantiation.

1. MLS MODEL: MLS security for single-level base relations.

SeaView TCB Security Model

A state is secure if it satisfies the STATE PROPERTIES.

A command is secure if, beginning in a secure state, it ends in a secure state AND satisfies all TRANSITION PROPERTIES.

A transaction is secure if it satisfies the SEQUENCE PROPERTIES.

SeaView SECURITY THEOREM: IF the initial state is secure,
AND all commands are secure,
AND all transactions are secure,
AND all axioms (state-independent properties) are satisfied
THEN the system is secure.

SeaView Model Properties

Classification constraints
Integrity constraints
Rule-based classifications

Discretionary properties

Constraints on multilevel relations
Multilevel integrity rules
Multilevel entity integrity
Multilevel referential integrity
Application-dependent contraints

Multilevel Survivability

Explore principles of the MLS/MLI/ MLA/MLX concepts to distributed systems and networks that must be survivable, encompassing security and reliability, within the framework of mechanisms providing generalized dependence.

Strive to avoid all of the pitfalls that have retarded the appearance of true MLS systems and networks, with realistic common sense, hindsight, and foresight.

Encryption

Good encryption must be used, securely embedded into survivable implementations. Compromises from within and below, bypasses, etc., must be addressed.

We need pervasive use of cryptographically based authentication (one-time passwords).

Robust key management is essential. Strong public-key crypto is recommended.

Key escrow/recovery seems appealing to law enforcement and intelligence, but is potentially disastrous to everyone else, including national security.

"Trusted Paths" and Resource Integrity

Without trustworthy paths to systems from users, to users from systems, and between systems, all bets are off.

Trusted paths are difficult in inherently untrustworthy systems.

The University of Pennsylvania Secure Bootstrap represents an attempt to overcome PC untrustworthiness.

Independent of MLI concepts, integrity checks are desirable for critical entities.

Once-Writable Virtual Store

Greatly simplifies historical database maintenance, inegrity, tamper resistance, secure audit recording, recovery, backup. (Cf. the Plan 9 file system)

Lower-level overwritable store hidden (encapsulated).

Used in Postgres (cf. Ingres)

Trustworthy Servers

- -

File servers, network servers, name servers, authentication servers, real-time analysis are all vital.

Servers must be survivable, and especially secure and reliable.

Sound operating systems, networking, and encryption are absolutely essential for servers, even more than on end-user systems.

The Mobile-Code Paradigm

- -

Write-once, (verify-once,) run-anywhere: potential for greater survivability. WOVORA: verify, add integrity seals.

Survivable mobile code requires secure operating systems and networking, sensible application design, good software engineering, and pervasive attention to implementation detail. Encryption, digital signatures, confined execution, proof-carrying code, formal methods helpful.

Need to reduce the necessary dependence on untrustworthy components, invoking generalize dependence mechanisms. The MLX concept is useful in structuring.

Mobile-Code Architectures

- -

A significant new paradigm for controlled execution involves the use of mobile code, that is, code that can be executed independently of where it is stored. The most common case involves portable reusable code that is acquired from some particular sources (remote or local) and executed locally. From a different perspective, it could involve local code that is executed elsewhere, or remote code that is executed at another remote site. Ideally, mobile code should be platform independent, and capable of running anywhere irrespective of how and from where it was obtained. Used in connection with the principles of separation of domains and allocation of least privilege, dynamic linking and dynamic loading with persistent access controls, this paradigm provides an opportunity for the secure execution of mobile code, and represents a very promising approach for achieving ultrasurvivable systems.

Of course, you can have major integrity, confidentiality, availability, denial-of-service, and general survivability risks involved in executing arbitrary code on one of *your* systems. The existence of mobile code whose origin and execution characteristics are typically not well known necessitates the enforcement of strict security controls to prevent Trojan horses and other unanticipated effects. It may be necessary to provide repeated reauthentication and validation, and revocation or cache deletion as needed. When combined with digital signatures and proof-carrying code to ensure authenticity and provenance, dynamically-linked mobile code provides a compelling organizing principle for highly survivable systems.

In principle, properly implemented environments for executing mobile code can contribute to survivability in various ways:

- Enable the execution of machine-independent trustworthy programs that have been carefully analyzed. Thus the paradigm becomes not just write-once run-anywhere (WORA), but rather write-once, verify-once, and run-anywhere (WOVORA). This can greatly enhance survivability by reducing the otherwise enormous task of verifying many different versions of the same code.

- Enable a distinction between the execution of programs on intrinsically unreliable and unsecure sites from execution on reliable and secure sites. In the sense of multilevel survivability, critical operations should not depend on code or execution on less trustworthy sites. In order for an MLX concept to be enforced, some measure of subsystem survivability must exist from which the aggregate survivability can be inferred or derived.

A highly survivable overall mobile-code architecture can be aided by a combination of trustworthy servers, encrypted network traffic, digital signatures, proof-carrying code, and other components and concepts. (See Sections 8.3 and 8.4 of the arl-one report.)

Three contemporary doctoral theses provide important contributions to the establishment of such an architecture, the formal-methods language-centric considerations of Drew Dean, the system-protection-centric work of Dan Wallach, and the proof-carrying code approach of George Necula. Background on understanding code mobility rather independently of survivability and security issues is given in a useful article by Fuggetta et al. (in a special issue of the *IEEE Transactions on Software Engineering* on mobility and network-aware computing). Formal methods are also particularly relevant to mobile code, because of the critical dependence on type safety (for example, the formalization of dynamic and static type checking for mobile code given by Riely and Hennessy).

An extraordinary compilation of articles on various aspects of the mobile code paradigm has been assembled by Giovanni Vigna, and published by Springer Verlag. This book reflects most of the potential problems with mobile code, and suggests numerous approaches to reducing the risks. Considering the enormous potential impact, this book is mandatory reading for anyone trying to use the mobile-code paradigm in supposedly survivable systems. The book also contains copious references, summarized in the arl-one report.

Emerging Trends in Architecture

- -

Patchworked systems and networks, networked PCs, borrowed software, full of vulnerabilities.

Hardware becoming cheap, fast, powerful, small; development of prototype chips as well as production becoming automated

Software development still costly, methodological aids and tools useful, formal methods becoming realistic

Lots of useful research in distributed systems and control, system architecture, authentication, knowledge-based interfaces, ...

Network requires a pervasive system view!

Reading for the Next Class Period

- -

This paper on anomaly and misuse detection is particularly relevant because of its emphasis on the importance of architecture and good software engineering:

P.G. Neumann and P.A. Porras, Experience with EMERALD to Date, Proceedings of the First USENIX Workshop on Intrusion Detection and Network Monitoring, Santa Clara, California, April 1999, pages 73–80:
http://www.csl.sri.com/neumann/det.html

A paper referred to therein is also on-line:
http://www2.csl.sri.com/emerald/emerald-niss97.html

I will use some slides that are not included here, most of which can be found at
http://www.csl.sri.com/intrusion/
(click on EMERALD).

Architectures for Survivability 2

ENPM 808s
Information Systems Survivability:
9. Architectures for Survivability 2

- -

The importance of human interfaces; various open-source paradigms; real-time monitoring of survivability, including anomaly and misuse detection covering penetrators and insiders

Use of Open-Source Software

Because some of the serious systemic deficiencies are not likely to be overcome in proprietary systems, it would be advantageous to make more systematic use of nonproprietary open-source software, especially if trustworthy distribution paths can be used consistently. Dependence on proprietary code can be seriously complicated any of a variety of factors:

- Size, which in some cases can be enormous (e.g., Windows NT)
- Inscrutability, which means it is practically unanalyzable
- Unavailability of source code, which means it cannot be readily adapted to different

situations or easily repaired when seriously flawed

It is a sad commentary on commercial and proprietary software development that some of the most useful software components are open-source software products that are the results of labors of love, and widely available free of charge over the Internet, such as the Emacs editor and the LaTeX document system (both of which have been used in the preparation of this report), Perl (Practical Extraction Resource Language), Bind (allowing symbolic naming of IP addresses), the Netscape browser source code, the Apache Web server, Berkeley BSD Unix, Linux, to name just a few. It is unfortunate that the same is not more widely true of security systems, although the Diffie-Hellman crypto algorithm is now in the public domain and a few simple schemes such as S-Key one-time passwords are freely available. Similarly,

PGP (Pretty Good Privacy) is becoming more widespread as it becomes seamlessly embedded in e-mail environments. Although there are risks that Trojan horses might be implanted in variant versions of open-source software, trustworthy software distribution is using public-key authentication schemes can overcome some of the uncertainty.

Real-time Analysis of Behavior and Response

There is a great need for the ability to provide real-time detection and analysis of system and network behavior, with appropriate real-time responses, from the coordinated perspective of survivability and its subtended requirements. There has been considerable work on this topic for almost two decades.

SRI has pioneered work on rule-based expert system analysis and statistical analysis, through IDES (Intrusion Detection Expert System) and NIDES (Next-Generation IDES). The current work on EMERALD (Event Monitoring Enabling Responses to Anomalous Live Disturbances) is the current extension of IDES and NIDES to monitor network activity. Overall, there are no

efforts that we know of besides EMERALD that are oriented toward the ability to detect problems arising in connection with generalized survivability. (See http://www.csl.sri.com/intrusion.html.)

There are many other institutions that have been developing systems addressing various aspects of the intrusion-detection problem, typically using either rule-based techniques or statistical analyses, but in most cases not both, and usually dealing with users of individual systems or local networks. See arl-one for copious references.

Anomaly and Misuse Detection

“Intrusion Detection” is too narrow a term, because we need to encompass insider misuse, and threats to reliability and survivability. “Misuse and anomaly detection” is somewhat closer to what is needed, and should encompass insiders and outsiders, intentional and accidental misuse, and ultimately reliability and survivability issues.

Tremendous reliance on these technologies are being recommended by various Government organizations. However, much greater emphasis on robustness and prevention of misuse is needed.

Commercial systems are relatively inflexible, heavily slanted toward exploitations of known vulnerabilities by intruders, using string matching or pattern matching,

largely ignoring insider attacks, reliability vulnerabilities, and hitherto unrecognized threats. For a recent workshop on insider misuse, see http://www.csl2.sri.com/insider-misuse/

Detection can be aided by better prevention.

Real-Time Analysis for Survivability: Event Monitoring Enabling Responses to Anomalous Live Disturbances (EMERALD)
http://www.csl.sri.com/intrusion.html and
http://www.csl.sri.com/neumann/det99.html
Numerous slides can be found at the former URL.

- -

Distributed real-time analysis, with various analytic engines: expert system seeks known attack signatures, statistics seek deviations from expected normal behavior, capable of enterprise-wide hierarchical correlation

Well software engineered for generality, flexibility, scalability, interoperability with other systems, rapid lightweight deployment, polymorphic use via a library of application bases for different targets

Applicable to heterogeneous events (systems, servers, network services) and requirements (survivability, security, reliability, etc.)

"Motherhood" is Short Shrifted!

- -

Functional requirements for survivability and its subtended abilities must be correct, consistent, complete, and precisely specified from the outset. Requirements must also anticipate operational needs such as interoperability, reusability, and evolvability.

Architectural fundamentals must be sound. Designs must be thoroughly specified, and evaluated for consistency with their requirements prior to implementation.

Implementation strategy must be realistic, adhered to consistently, and demonstrably capable of satisfying the requirements.

Plans for testing must be anticipatory and thorough.

A Little Wisdom

- -

Understand your long-term goals, as well as the threats and risks.

Understand your operational needs up front.

Understand the practical limitations of your vendors, developers, administrators, and operational personnel.

Good software engineering practice and foresight are incredibly valuable, and worth the extra effort in complex systems and networks.

Short-term optimization is usually counterproductive.

A Baseline Survivable Architecture

- -

Future work will include the definition and detailed documentation of a baseline family of architectures capable of satisfying critical survivability requirements: an evolutionary open-system approach, minimizing the scope of trustworthiness, including trusted paths where needed, systemic authentication, robust encryption, demonstrably trustworthy mobile code, generalized-dependence mechanisms as needed, accommodating MLS/MLI/MLX as needed without unduly complicating single-level operation, extensive real-time anomaly/misuse detection, ... [From arl-one]

Putting It All Together

- -

Establish the mission requirements and map them into specific system/network requirements.

Establish a specific instance of the baseline architecture.

Analyze it with respect to the requirements.

Refine the architecture and document it.

Implement it, test it, red-team it, operate it, inject faults and attacks, and determine the effects, let the users in and see if it prevents penetrations and other misuse, iterating whenever necessary. Be wary rather than keeping your head in the sand.

Residual Vulnerabilities and Risks

Despite your very best efforts, there will still be vulnerabilities, serious threats that are unchecked, and risks:

Hardware failures
Software flaws
Malicious penetrations
Trojan horses
Insider misuse
Inadvertent human actions
Environmental disasters
Other operational hazards
Other unforeseen circumstances!

Anticipate these realities, and plan accordingly. Misuse and anomaly detection remains desirable, even in the very best of circumstances.

Some Suggested Future R&D Directions

Establish a more detailed set of generic system/network requirements for survivability and characterize their interdependencies.

Establish baseline families of survivable architectures and document them thoroughly.

Define more-robust networking protocols

Establish a survivable foundation for mobile code, specifying requirements on operating systems, browsers and servers, boundary protection, network protocols, programming languages, system composition, and other factors, and defining adequate system/network architectures

Develop prototype testbeds that realize the best architectures, and conduct experiments.

Develop a rigorous framework for generalized dependence

Formalize predictable subsystem composition

Formalize the process of transforming “mission requirements” into specific requirements (subset of generic)

Provide a rigorous basis for trustworthiness-preserving dynamic system reconfiguration and dynamic adaptability

PCCIP Organizational Recommendations

Office of National Infrastructure Assurance
National Infrastructure Assurance Council
Infrastructure Assurance Support Office
A Federal Lead Agency for each sector
Sector Infrastr. Assurance Coordinator
Information Sharing and Analysis Center
Warning Center

Presidential Decision Directive 63 transformed the PCCIP into the Critical Infrastructure Assurance Office (CIAO), to support the National Coordinator (overseeing critical infrastructure protection and foreign terrorism).

Information survivability and its subtended requirements must be addressed fully.

PCCIP R&D Recommendations

1. Information assurance
2. Intrusion monitoring and detection
3. Vulnerability assessment and systems analysis
4. Risk-management decision support (risky?)
5. Protection and mitigation
6. Incident response and recovery

The recommendations here extend security to survivability, and address items 1, 2, 3, 5, and the recovery part of 6. The incident response part is useful, but has serious effectiveness problems. Item 4 is risky (see Computer-Related Risks, pp.255-257)!

Education and Training

Concurrently with R&D efforts, we recommend preparing detailed course materials inspired by this project, teaching those materials in institutional settings (government, university, corporate), and influencing the integration of survivability concepts into regular curricula.

A strong understanding of the requirements for survivable systems and networks, the vulnerabilities, threats and risks, the deficiencies in available (sub)systems, necessities for robust architectures, and guidelines for system development (and in the case of DoD and corporations, for procurement) is essential for everyone involved in systems intended to be survivable.

Reliability in Perspective

ENPM 808s
Information Systems Survivability:
10. Reliability in Perspective

- -

Fault tolerance, other constructive uses of redundancy, robust mobile code, integration of security and reliability

Different degrees of fault tolerance, fault detection, fault avoidance, dynamic fault elimination, fault tolerance (e.g., fail-safe, fail-soft, fail-fast, fail-secure), fault recovery (forward or backward), self-checking components, ...

This lecture is in memory of my mentor, colleague, and friend, David Huffman, who died on October 7 1999.

Fault Tolerance

- -

There is a vast literature of techniques for fault tolerance that this report does not attempt to replicate. Copious references are given in the arl-one report.

For example, techniques for increasing system reliability in response to hardware faults and communications failures are explored in general.

Once again demonstrating the desirability of a confluence of requirements and a corresponding confluence of techniques for combatting security and reliability problems along the lines of the reconvergence of availability requirements in the figure, consider the requirements for data integrity in the sense of no-unintended-change shown at the nodes designated by a sharp (#) in

the figure. Data integrity can be enhanced through cryptographic integrity checks (typically to protect against malicious alterations) or error-correcting coding techniques (typically to protect against accidental garbling). However, an interesting recent special-purpose use of coding for detecting malicious tampering as well as accidental errors in once-writable optical disks is given by Blaum et al., taking advantage of the asymmetry inherent in certain once-writable storage media in which writing can only change the state of a bit in one direction (e.g., from a not previously written *zero* bit value to a written *one* bit, but never the reverse). This is another example of a cross-over implementation that can simultaneously address different sets of subrequirements stemming from otherwise independent-seeming major requirements. In such cases, considerable benefits can be obtained by recognizing the commonality among otherwise independent subrequirements and then providing a unified treatment in the design and implementation.

```
 Error Correcting Codes

Hamming Distance represents bit changes

 0      1      2      3      4      5
 00000--00001--00011--00111--01111--11111

Minimum   Correction/
Distance  Detection
--------  -----------------------
   1      No detection
   2      SINGLE-error DETECTION
   3      SINGLE-error CORRECTION
            OR DOUBLE-error DETECTION
   4      SINGLE-error CORRECTION *AND*
          DOUBLE-error DETECTION,
            OR TRIPLE-error DETECTION with
               NO error correction
   5      DOUBLE-error CORRECTION,
            OR SINGLE error CORRECTION AND
               DOUBLE-error DETECTION
            OR QUADRUPLE-error DETECTION
  etc.
```

```
Huffman's View of the Hamming Code
     ----------
    /          \
   /            \
  /            --\------
 /      1     /   \     \
|           /  3  |     \
|     ------/--   |      \
|    /    /   \   |       \
 \  /     |    \ /         |
  \/   5  |  7  \/    2    |
  /\      |     /\         |
 |  \      \   /  |       /
 |   --------\--  |      /
 |            \ 6 |     /
  \     4      \ /     /
   \           -/------
    \          /
     \        /
      ---------
```



```
Hamming single-error correcting code
Regions are bit positions ($n=7$)
Circles are check sums ($r=3$),
and thus r check bits (bits 1,2,4).
Others ($k=n-r=4$) are info bits (3,5,6,7)
```

```
Huffman's View of Hamming Code
(n, k, d) = (7, 4, 3)

Parity    Info       Error
 bits     bits       syndromes
 1 2 4   3 5 6 7    ------------
 -----   -------    check:4 2 1
 0 0 0   0 0 0 0    ----\ 5 3 3
 1 1 1   0 0 0 1    bad | 6 6 5
 0 1 1   0 0 1 0    bit | 7 7 7
 1 0 0   0 0 1 1    -----------
 1 0 1   0 1 0 0    none| 0 0 0
 0 1 0   0 1 0 1      1 | 0 0 1
 1 1 0   0 1 1 0      2 | 0 1 0
 0 0 1   0 1 1 1      3 | 0 1 1
 1 1 0   1 0 0 0      4 | 1 0 0
 0 0 1   1 0 0 1      5 | 1 0 1
 1 0 1   1 0 1 0      6 | 1 1 0
 0 1 0   1 0 1 1      7 | 1 1 0
 0 1 1   1 1 0 0
 1 0 0   1 1 0 1
 0 0 0   1 1 1 0
 1 1 1   1 1 1 1
```



```
Huffman Graph Codes

    6       Edges are bit positions
 :--------:   1,2,3,4,5,6 (n = 6)
 |\      /|  Redundant bits are
 | \1   / |   any max spanning tree
 |  \  /  |   such as 1,2,3 (r = 3)
5|   \/   |3 Information check bits are the
 |   /\   |   others, 4,5,6 (k = n-r = 3)
 |  /  \  |  Check sums are any r INDEPENDENT
 | /2   \ |   loops, (1,4,5), (2,5,6), (3,4,5,6)
 |/      \|  The minimum Hamming distance of the
 :--------:   code is the MINIMUM LOOP length.
    4
```



```
    Code          Error Syndromes
--------------   -----------------
 1 2 3  4 5 6    check:   1  2  3
parity  info     ----\    4  5  4
bits    bits     bad |    5  6  5
--------------   bit |          6
 0 0 0  0 0 0    ----------------
 0 1 1  0 0 1    none|    0  0  0
 1 1 1  0 1 0     1  |    1  0  0
 1 0 0  0 1 1     2  |    0  1  0
 1 0 1  1 0 0     3  |    0  0  1
 1 1 0  1 0 1     4  |    1  0  1
 0 1 0  1 1 0     5  |    1  1  1
 0 0 1  1 1 1     6  |    0  1  1
                 ??? |    1  1  0
QUESTION: What error patterns could
have produced the syndrome 110?
HINT: There is more than one set of
such patterns.
```


Other Types of Redundant Coding

- -

Asymmetric errors (e.g., dropping ones only),

Correlated errors (e.g., bursts)

Arithmetic errors (e.g., in addition or multiplication)

Self-checking circuits (e.g., with syndromes for detection)

Some Other Uses of Redundancy

Alternative routing in networks

Alternative sources of replicated data, with various different forms of replication (synchronous write-through, asynchronous write-through, less recent versions, backup and restoration, etc.)

Alternative computation sites, with various forms of replication (identical replication as hot standby, or with real-time voting; nonidentical replication; degraded standby)

Majority Voting

In the SIFT architecture discussed in the lecture 8, significant increases in reliability (four orders of magnitude in mean-time to failure) were achieved through majority voting. The replicated cost in processors (and memories, buses, and power supplies, sevenfold) is small relative to the cost of the airplane.

However, if hardware is consistently flawed, or a software-implemented algorithm is inherently flawed, or the replicated code is buggy, then voting is worthless. Note that SIFT used formal methods to verify critical software modules.

N-Version Programming ("Program Diversity")

Various experiments have been conducted with majority voting among separate programs all written to the same specification. This approach has some serious drawbacks:

Common-mode failures (Leveson and Knight, *FTCS 16*, 1986; Brilliant, Knight, and Leveson, *IEEE Trans. Softw. Eng.,* November 1989 and February 1990; Shimeall and Leveson, IEEE TSE, February 1991)

One excellent programmer is likely to do better than three mediocre programmers.

Byzantine Faults

Byzantine fault mode: ARBITARY behavior.

A simple example of clock synchronization using three clocks broadcasting times to each other, each averaging among the three values. A and B are true clock sources (1130). C is Byzantine, e.g., maliciously misleading the others (sending different values to different neighbors).

	A->B	A->C	B->A	B->C	C->A	C->B	AVE
A			1130		1200		1145
B	1130					1100	1115
C		1130		1130			1130

Note: averaging clock algorithms are risky. The old 11-clock ARPAnet algorithm discarded the two most extreme values, but still got into trouble occasionally.

Byzantine Agreement

Simple example: Four clocks ($4 = 3x1 + 1$) (Lamport, Shostak, Pease, ACM *TOPLAS,* July 1982 based on earlier 1980 paper)

First round: Each processor broadcasts its own private values.

Second round: Each processor broadcasts all private values from in the first round.

Each processor then compares the received private values of the other clocks. If the majority of those agree, that private value is accepted. You can work it out, or read the paper. In general, $n = 3k + 1$ requires k+1 rounds, and tolerates k arbitrarily Byzantine components. Many subsequent papers on Byzantine agreement.

Open Discussion Topics

We revisit the material on hierarchical layering of reliability and fault tolerance techniques (e.g., see Illustrative Reliability Threats, 61-63; Survivability Attributes at Different Layers, 74-75; Typical Defensive Measures, 97), and consider which of these techniques might be most useful. We also consider which techniques might have helped overcome some of the cases considered earlier.

Cases Discussed Previously

- -

1980 ARPAnet collapse: error detection in memory; robust garbage collection algorithm;

Yorktown Divide-by-Zero: argument checking

Western power outage propagation: greater redundancy in resources and in control?

1997 Kansas City power outage: self-checking?

Black Hawk helicopter and many other EMI problems: shielding!

1990 AT&T collapse and 1998 AT&T packet-data frame-relay network collapse: better testing and configuration control on software upgrade?

Other cases

Security in Perspective

ENPM 808s
Information Systems Survivability:
11. Security in Perspective

- -

System and network security, trustworthiness, preventing denials of service (more broadly than just defending against attacks), roles of encryption, secure mobile code.

[UMd Professor Virgil Gligor joined me, with both of us in person at U Md for this lecture. Virgil has an incredibly broad range of experience relating to security, and this resulted in some free-wheeling open discussion on important issues related to survivability, including preventing denials of service, system architecture, the relevance of theoretical contributions, the importance of sound practice, etc.]

Open Discussion Topics

- -

Security requirements: noncompromisibility, networking, interoperability; defense against insiders, outsiders, denials of service, EMI, ..., according to required trustworthiness
Evaluation criteria, including for networking
Architectures, domains, boundary control, insiders, minimized trustworthiness, secure mobile code, Trusted Xenix, partitioned root privileges, malicious code, Y2K remediation problems
Relevance of MLS, kernels, TCBs,
Good cryptography, properly embedded
Dependability of generalized dependence
Role of software engineering, open source, education, IETF, IPSEC, ICANN, and lots more ...

Architectures for Survivability 3

ENPM 808s
Information Systems Survivability:
12. Architectures for Survivability 3

- -

Integrating the requirements and the components seamlessly, interoperably, predictably, more on the thin-client distributed selective trustworthiness architecture, etc., assessing the big picture

What is the role of architecture in all of the Illustrative Risks cases? Open discussion.

Talk on a student project

Implementation Considerations

ENPM 808s
Information Systems Survivability:
13. Implementation Considerations

- -

Implementing and configuring for survivability: putting it all together into a coherent approach to survivable systems and networks

Recommended research directions NOTE: Final projects are due by the final class period.

A Few Remarks to Reflect On

- -

There are no easy answers. It is a long way from theory to practice, from concept to workable implementations, from research to development. What is commercially available is mostly sorely lacking with respect to what is needed. Today's operating systems, Internet protocols, and networking are inadequate in the context of survivability requirements.

Building robust systems out of less robust components must become more viable. However, we must remember that completely flaky infrastructures may allow compromises. Thus, some trustworthy components are essential.

Generalized dependence works well for fault tolerance and pure reliability applications

such as mean-time to failure. For security, wrappers, firewalls, TCBs, k/n crypto, and Byzantine security algorithms are still subject to compromises (from within and from below).

We need fully fledged prototypes that address the realistic requirements and that use the best research results. The trustworthy servers of the thin-client architecture are an important example.

We must have practical systems for real-time analysis and response. (EMERALD is an outstanding example!)

Survivability has widespread threats. We need education and awareness of government folks, developers, professors, students, media moguls, and citizens...

Simplicity is an evasive goal, especially when we are dealing with inherently complex systems that must work within inherently complex environments involving fallible or culpable people. Vitualizing simplicity makes good sense, using abstraction, encapsulation, layering, perhaps object orientation, and generally good software engineering practice.

People will always be a weak link. Good technology is vital, but do not overendow it.

Needs for the Future

- -

Generic requirements encompassing all of the potentially critical subrequirements for survivability, which can then be instantiated for specific applications

Much more robust network and distributed system protocols. [In the lecture, I noted that there is a Y2K-like problem with IPSEC: IP version 16!!! One answer is to wrap around with a windowed approach, assuming the old versions are no longer around, although that might get into lawsuits with the guy who is suing the Y2K folks who used that technique.]

Libraries of demonstrably sound robust procedures for generalized dependence

Open-system and nonproprietary open-source architectural components compatible with

survivability requirements, with interoperability, ready composability, trustworthy components as needed, trustworthy distribution, ... Possible impetus on commercial vendors?

Sound cryptographic infrastructures for authentication, integrity, confidentiality

Still much research and prototype development to be done.

There are huge educational problems, to which this course may be just one step.

Important Research Directions

- -

Generic requirements
Robust protocols
Detailed open-system architectures and support for robust mobile code, predictable composability, trustworthy critical components
Establish reusable constructs for generalized dependence
Interoperable systems for real-time monitoring of all survivability-relevant attributes
Predictably controllable and constrained dynamic reconfiguration and dynamic system adaptability

Fundamental Lessons of History

- -

Requirements must be sufficiently correct, consistent, complete, and precisely specified, from the outset, encompassing functional and operational requirements.

Proactive architectures are vital from the outset, as are software engineering, control and discipline in the development process.

Plans for testing, validation, verification, and acceptance should be thought out carefully in advance as well.

Requirements, design, implementation, and operational practice must anticipate iteration, evolution, and maintenance.

Specific Lessons

(Motherhood is easy, unless you are a mother.)

- -

Identify your long-range goals up front.

Fred Brooks' "Build one to throw one away" is usually impractical. Planning for evolution is vital, but difficult.

Good software engineering practice and good system architectures are invaluable.

Short-term and local optimization are very risky.

A Few More Comments

- -

This is not a case of Just Add Money and Stir. There is a long way to go.

Serious understanding is needed of the depths of the problems and the urgent needs for solutions.

Although progress can be made in the short term, commitment to long-term advances is essential.

Despite all of this, there will still be residual risks. [DISCUSSION]

Final Class

ENPM 808s
Information Systems Survivability:
14. Conclusions
- -

Representative Discussion Topics for the Final Class
- -

The final class period includes considerable open discussion, providing an overview of the most important issues raised by the course, an assessment of where we need to go from here, and how to get there.

Topics include roles of formal methods, testing, more on architectural alternatives, what is needed to make open-source robustification realistic, residual risks, lessons learned, etc. Hopes for the future

[As it turned out, the first third of the last lecture was devoted to a discussion of the new air-traffic control system concept of “free flight”, with many of the existing controller functions being relocated into the cockpits. This introduces much more stringent security requirements (including nondenial of service) into the mix, and creates a highly distributed system that is highly relevant for this course.

Some Discussion Topics
- -

In the recent National Information Systems Security Conference in Crystal City, 18 October 1999, I chaired a panel in which Virgil Gligor, George Dinolt, Sami Saydjari, and Brian Snow participated. These are some of the discussion topics that I had suggested we consider relating to the realistic challenges of achieving dependably secure systems and networks. Appropriate perspectives include computer science education, software engineering, criteria, formal methods, multilevel security, government funding, market forces, and the open-source movement. This was not a hodge-podge of unrelated ideas, but rather an attempt to see if we can integrate all of these perspectives into a unified whole that is much greater than the sum of its parts. What lessons must be learned from our past experience, and what must we do differently in the future?

* What should our universities be teaching to increase the understanding of software development and dependably secure systems?

* What about training computer system administrators and system development managers?

* What are realistic expectations of software engineering disciplines in developing secure systems?

* What could the computer industry do differently that would be effective, within their needs for economic competitiveness?

* What R&D directions are not being adequately pursued that might realistically have a constructive effect on commercial systems and on government deployments?

* What can governments do differently that would encourage the availability of systems and networks with greater security?

* Are there relevant lessons that we should learn from the Y2K fiasco?

* What hope do we have for establishing criteria that meaningfully constrain security and that can be extensively fulfilled?

* Can the open-source, free software, and nonproprietary movements lead to systems that are dramatically more robust than commercial systems?

* Could serious liability laws constrain developers and purveyors of systems that engender huge losses or cause critical failures?

* To what extent are our critical national infrastructures affected by a lack of robust systems and networks?

Final Comment

- -

FROM arl-one: We have outlined many concepts that are highly relevant to the specification, design, development, and operation of highly survivable systems and networks. The architectural directions pursued here integrate all of those concepts and provide a strong basis for systems that accommodate mobile code, portable user platforms and robust execution platforms, minimal critical dependence on untrustworthy components, and highly reconfigurable and adaptive environments. There is much work yet to be done to demonstrate the applicability of this approach, but we hope that we have broken some new ground.