Life-Cycle Models for Survivable Systems

Richard C. Linger
Howard F. Lipson
John McHugh
Nancy R. Mead
Carol A. Sledge

October 2002

TECHNICAL REPORT
CMU/SEI-2002-TR-026
ESC-TR-2002-026

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

Life-Cycle Models for Survivable Systems

CMU/SEI-2002-TR-026
ESC-TR-2002-026

Richard C. Linger
Howard F. Lipson
John McHugh
Nancy R. Mead
Carol A. Sledge

October 2002

Networked Systems Survivability Program

Unlimited distribution subject to the copyright.

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Norton L. Compton, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2002 by Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

Acknowledgments vii

Abstract ix

1 Survivability and the System Life Cycle 1

2 Survivability Concepts 3
- 2.1 The New Network Paradigm: Organizational Integration 3
- 2.2 The Definition of Survivability 4
- 2.3 Characteristics of Survivable Systems 5
- 2.4 Survivability as an Integrated Engineering Framework 9

3 System Development Life-Cycle Models 11
- 3.1 The Spiral Model 11
- 3.2 A Spiral Model for Survivable Systems Development 13

4 System Development Life-Cycle Activities and Survivability 17
- 4.1 Requirements and Specification 17
  - 4.1.1 Expressing Survivability Requirements 18
  - 4.1.2 Requirements Definition for Essential Services 22
  - 4.1.3 Requirements Definition for Survivability Services 23
- 4.2 Architecture and Design 26
- 4.3 Implementation and Verification 29
  - 4.3.1 Defensive Coding Strategies 29
  - 4.3.2 Correctness Verification 31
- 4.4 Testing 32
  - 4.4.1 Penetration Testing 32
  - 4.4.2 Statistical Usage-Based Testing 33
- 4.5 System Evolution 33

5 COTS Development Life-Cycle Activities 39

6 COTS Development Life-Cycle Activities and Survivability 45
- 6.1 System Context Survivability Issues 48

7 Future Research Opportunities.. 51

References.. 53

List of Figures

Figure 1: A Project Spiral Cycle 13

Figure 2: Specialization of the Spiral Model for Survivability Driver 14

Figure 3: Classes of Requirements for Survivable Systems 19

Figure 4: Integrating Survivability into System Requirements 20

Figure 5: Architectural Level of a Survivable Network Design Method 27

Figure 6: Spiral Life-Cycle Model with Survivability Activities 40

List of Tables

Table 1: Key Properties of Survivable Systems..7

Table 2: Life-Cycle Activities and Corresponding Survivability Elements.............17

Table 3: Correctness Conditions for Functional Verification 32

Table 4: Trigger Elements for Evolutionary-Design Activities for Survivable Systems 35

Table 5: Possible Evolutionary-Design Activities in Response to a Trigger Event...37

Table 6: CBS Life-Cycle Activities 41

Table 7: COTS Life-Cycle Activities Tailored to Survivability 46

Acknowledgments

The authors are grateful for the excellent editorial support provided by Pamela Curtis.

Abstract

Today's large-scale, highly distributed, networked systems improve the efficiency and effectiveness of organizations by permitting whole new levels of organizational integration. However, such integration is accompanied by elevated risks of intrusion and compromise. Incorporating survivability capabilities into an organization's systems can mitigate these risks. Current software development life-cycle models are not focused on creating survivable systems, and exhibit shortcomings when the goal is to develop systems with a high degree of assurance of survivability. If addressed at all, survivability issues are often relegated to a separate thread of project activity, with the result that survivability is treated as an add-on property. For each life-cycle activity, survivability goals should be addressed, and methods to ensure survivability incorporated.

This report explains survivability concepts, describes a software development life-cycle model for survivability, and illustrates techniques that can be applied during new development activities to support survivability goals. It also describes a software life-cycle model and associated activities to support survivability goals for systems based on commercial off-the-shelf products.

1 Survivability and the System Life Cycle

Today's large-scale, highly distributed networked systems improve the efficiency and effectiveness of organizations by permitting whole new levels of organizational integration. However, such integration is accompanied by elevated risks of intrusion and compromise. Incorporating survivability capabilities into an organization's systems can mitigate these risks. As an emerging discipline, survivability builds on related fields of study (e.g., security, fault tolerance, safety, reliability, reuse, performance, verification, and testing) and introduces new concepts and principles. Survivability focuses on preserving essential services, even when systems are penetrated and compromised [Anderson 97].

Current software development life-cycle models are not focused on creating survivable systems, and exhibit shortcomings when the goal is to develop systems with a high degree of assurance of survivability [Marmor-Squires 88]. If addressed at all, survivability issues are often relegated to a separate thread of project activity, with the result that survivability is treated as an add-on property. This isolation of survivability considerations from primary system-development tasks results in an unfortunate separation of concerns. Survivability should be integrated and treated on a par with other system properties, to develop systems with required functionality and performance that can also withstand failures and compromises. Important design decisions and tradeoffs become more difficult when survivability is not integrated into the primary development life cycle. Separate threads of activities are expensive and labor intensive, often resulting in duplicated effort in design and documentation. In addition, tools for supporting survivability engineering are often not integrated into the software-development environment. With separate threads of activities, it becomes more difficult to adequately address the high-risk issues of survivability and consequences of failure. In addition, technologies that support survivability goals, such as formal specification, architecture tradeoff methods, intrusion analysis, and survivability design patterns, are not effectively applied in the development process.

For each life-cycle activity, survivability goals should be addressed, and methods to ensure survivability incorporated [Mead 00c]. In some cases, existing development methods can enhance survivability. Current research is creating new methods that can be applied; however, more research and experimentation are required before the goal of survivability can become a reality.

In this report, we describe survivability concepts, discuss a software development life-cycle model for survivability, and illustrate techniques that can be applied during new development

activities to support survivability goals. We also discuss a software life-cycle model and associated activities to support survivability goals for systems based on commercial off-the-shelf (COTS) software.

2 Survivability Concepts

Survivable systems research over the past few years has resulted in the development of the concepts and definitions of survivability described in this section. They are drawn from the work of the Survivable Network Technology Team at the Software Engineering Institute (SEI) and CERT® Coordination Center (CERT/CC) [Lipson 96, Ellison 99a].

2.1 The New Network Paradigm: Organizational Integration

From their modest beginnings some 20 years ago, computer networks have become a critical element of modern society. These networks not only have global reach; they also have impact on virtually every aspect of human endeavor. Network systems are principal enabling agents in business, industry, government, and defense. Major economic sectors, including defense, energy, transportation, telecommunications, manufacturing, financial services, health care, and education, depend on a vast array of networks operating on local, national, and global scales. This pervasive societal dependency on networks magnifies the consequences of intrusions, accidents, and failures, and amplifies the critical importance of ensuring network survivability.

As organizations seek to improve efficiency and competitiveness, a new network paradigm is emerging. Networks are being used to achieve radical new levels of organizational integration. This integration obliterates traditional organizational boundaries and transforms local operations into components of comprehensive, network-resident business processes. For example, commercial organizations are integrating operations with business units, suppliers, and customers through large-scale networks that enhance communication and services. These networks combine previously fragmented operations into coherent processes open to many organizational participants. This new paradigm represents a shift from bounded networks with central control to unbounded networks. Unbounded networks are characterized by distributed administrative control without central authority, limited visibility beyond the boundaries of local administration, and a lack of complete information about the network. At the same time, organizational dependencies on networks are increasing and risks and consequences of intrusions and compromises are amplified.

® CERT and CERT Coordination Center are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

2.2 The Definition of Survivability

We define *survivability* as the capability of a system to fulfill its mission, in a timely manner, in the presence of attacks, failures, or accidents. We use the term *system* in the broadest possible sense, including networks and large-scale systems of systems.

The term *mission* refers to a set of very high-level requirements or goals. Missions are not limited to military settings, since any successful organization or project must have a vision of its objectives whether expressed implicitly or as a formal mission statement. Judgments as to whether or not a mission has been successfully fulfilled are typically made in the context of external conditions that may affect achievement of that mission. For example, imagine that a financial system shuts down for 12 hours during a period of widespread power outages caused by a hurricane. If the system preserves the integrity and confidentially of its data and resumes its essential services after the period of environmental stress is over, the system can reasonably be judged to have fulfilled its mission. However, if the same system shuts down unexpectedly for 12 hours under normal conditions or minor environmental stress, thereby depriving its users of essential financial services, the system can reasonably be judged to have failed its mission, even if data integrity and confidentiality are preserved.

Timeliness is a critical factor that is typically included in (or implied by) the very high-level requirements that define a mission. However, timeliness is such an important factor that we included it explicitly in the definition of survivability.

The terms *attack*, *failure*, and *accident* are meant to include all potentially damaging events; but these terms do not partition these events into mutually exclusive or even distinguishable sets. It is often difficult to determine if a particular detrimental event is the result of a malicious attack, a failure of a component, or an accident. Even if the cause is eventually determined, the critical immediate response cannot depend on speculations about such future knowledge.

Attacks are potentially damaging events orchestrated by an intelligent adversary. Attacks include intrusions, probes, and denials of service. Moreover, the threat of an attack may have as severe an impact on a system as an actual occurrence. A system that assumes a defensive position because of the threat of an attack may reduce its functionality and divert resources to monitoring the environment and protecting system assets.

We include failures and accidents in the definition of survivability. *Failures* are potentially damaging events caused by deficiencies in the system or in an external element on which the system depends. Failures may be due to software design errors, hardware degradation, human errors, or corrupted data. *Accidents* describe a broad range of randomly occurring and potentially damaging events, such as natural disasters. We tend to think of accidents as externally generated events (i.e., outside the system) and failures as internally generated events.

With respect to system survivability, a distinction between a failure and an accident is less important than the impact of the event. Also, it is often possible to distinguish between intelligently orchestrated attacks and unintentional or randomly occurring detrimental events. Our approach concentrates on the effect of a potentially damaging event. Typically, for a system to survive, it must react to (and recover from) a damaging effect (e.g., the integrity of a database being compromised) long before the underlying cause is identified. In fact, the reaction and recovery must be successful whether or not the cause is ever determined.

The primary focus in this paper is to provide managers with methods to help systems survive the acts of intelligent adversaries. While the focus is on intrusions, the methods discussed apply in full measure to failures and accidents as well.

Finally, it is important to recognize that it is the mission fulfillment that must survive, not any particular subsystem or system component. Central to the notion of survivability is the capability of a system to fulfill its mission, even if significant portions of the system are damaged or destroyed. We use the term *survivable system* as a shorthand for a system with the capability to fulfill a specified mission in the face of attacks, failures, or accidents. Again, it is the mission, not a particular portion of the system that must survive.

2.3 Characteristics of Survivable Systems

A key characteristic of survivable systems is their capability to deliver essential services in the face of attack, failure, or accident. Central to the delivery of essential services is the capability of a system to maintain essential properties (i.e., specified levels of integrity, confidentiality, performance, and other quality attributes) in adverse environments. Thus, it is important to define minimum levels of such quality attributes that must be associated with essential services. For example, a launch of a missile by a defensive system is no longer effective if the system performance is slowed to the point that the target is out of range before the system can launch.

These quality attributes are so important that definitions of survivability are often expressed in terms of maintaining a balance among multiple quality attributes, such as performance, security, reliability, availability, fault-tolerance, modifiability, and affordability. The Architecture Tradeoff Analysis project at the SEI is using this attribute-balancing (i.e., tradeoff) view of survivability to evaluate and synthesize survivable systems [Kazman 98]. Quality attributes represent broad categories of related requirements, so a quality attribute may be composed of other quality attributes. For example, the security attribute traditionally includes three subattributes, namely, confidentiality, integrity, and availability.

The capability to deliver essential services, and maintain associated essential properties, must be sustained even if a significant portion of a system is incapacitated. Furthermore, this capa-

bility should not be dependent upon the survival of a specific information resource, computation, or communication link. In a military setting, *essential services* might be those required to maintain an overwhelming technical superiority, and *essential properties* may include integrity, confidentiality, and a level of performance sufficient to deliver results in less than one decision cycle of the enemy. In the public sector, a survivable financial system might be one that maintains the integrity, confidentiality, and availability of essential information and financial services, even if particular nodes or communication links are incapacitated through intrusion or accident, and that recovers compromised information and services in a timely manner. The financial system's survivability might be judged by using a composite measure of the disruption of stock trades or bank transactions (i.e., a measure of the disruption of essential services).

Key to the concept of survivability, then, is identifying the essential services (and the essential properties that support them) within an operational system. Essential services are defined as the functions of the system that must be maintained when the environment is hostile or when failures or accidents occur that threaten the system. To maintain their capabilities to deliver essential services, survivable systems must exhibit the four key properties illustrated in Table 1, namely, *resistance, recognition, recovery* (the three R's), and *adaptation*.

The table identifies a number of *survivability strategies* that can be applied to counter threats of an overt attack on a system. Some of these techniques for enhancing survivability are borrowed from other areas, notably the security, safety, and fault-tolerance communities.

In the area of attack resistance, a number of techniques are available. User authentication mechanisms limit access to a system to a group of approved users. Authentication mechanisms range from simple passwords to combinations of passwords, user-carried authentication tokens (themselves password protected), and biometrics. Access controls can be applied to system access or to individual programs and data sets. Access controls, enforced by a trustworthy operating system, automatically apply a predefined policy to grant or deny access to an authenticated user. When properly used and implemented, access controls can serve as a substitute for program- and data-set-level password mechanisms.

Table 1: Key Properties of Survivable Systems

Key Property	Description	Example Strategies
Resistance to attacks	Strategies for repelling attacks	Authentication Access controls Encryption Message filtering Survivability wrappers System diversification Functional isolation
Recognition of attacks and damage	Strategies for detecting attacks and evaluating damage	Intrusion detection Integrity checking
Recovery of essential and full services after attack	Strategies for limiting damage, restoring compromised information or functionality, maintaining or restoring essential services within mission time constraints, restoring full services	Redundant components Data replication System backup and restoration Contingency planning
Adaptation and evolution to reduce effectiveness of future attacks	Strategies for improving system survivability based on knowledge gained from intrusions	New intrusion recognition patterns

Encryption can protect data, either within a system or in transit between systems, from interception or physical capture. Available encryption technologies are strong enough to resist all currently feasible brute-force attacks. Encryption translates the problem of protecting large quantities of data into a problem of managing relatively small quantities of keying material. Encryption can also be used to provide authentication, non-repudiation, integrity checking, and a variety of other assurance properties.

Message filtering is typically used at the boundary of a system or installation to restrict the traffic that enters the system. For example, there is no reason to allow messages related to unsupported or unwanted services to enter an installation. Messages appearing to originate from within an installation are probably not legitimate if coming from the outside, and messages that appear to originate outside should not be let out. Filters can be designed to block messages associated with known attacks, as well.

Survivability wrappers are essentially message filters applied at the OS interface level. They may be used to provide operand checking or to redirect calls to unsafe library routines to more robust versions. They may also be used to impose a restrictive access-control policy on a particular application. System diversification combined with redundant implementations makes an attacker's job more difficult. In a diverse implementation, it is likely that a scenario used to attack one implementation will fail on others. Defensive coding is used to protect programs from bad input values. This technique is discussed in more detail in Section 4.3.1.

Functional isolation reduces or eliminates dependencies among services to the greatest extent possible. This also prevents an attack on one service from compromising others. Isolation is often not easy to achieve, as dependencies among services are not obvious if viewed at the wrong level of abstraction. Services that share a processor, for example, are mutually dependent on one another for CPU and memory resources. They may also share disk space and probably a network adapter. It is possible for one process to launch a denial of service attack on another by gaining a monopoly on any of these resources. Resource isolation may require a quota-based sharing mechanism or similar technique. Functional isolation can extend to physically separating system functions, often on separate servers with no logical connections—for example, separating email processing from sensitive data files. No electronic intrusion method can jump an air gap or penetrate a machine that is powered down.

In the area of attack recognition, there is a limited number of choices. Intrusion detection systems typically attempt to identify attacks by either looking for evidence of known attack patterns or by using a baseline model of normal system behavior to treat departures from normal as potential attacks. Both techniques can be applied to network traffic as well as to platform- or application-specific data. System auditing and application logs are sources of information for detecting intrusions at the platform or application level. Both real-time and post-processing intrusion-detection systems (IDSs) exist. At the present time, IDSs miss many intrusions, especially new or novel attacks, and suffer from high false-alarm rates. Integrity checkers can detect intrusions that modify system files or data that should remain unchanged. The checking process involves creating a baseline model of the files to be protected using checksums or cryptographic signatures, and periodically comparing the current model to the baseline.

In terms of recovery, when a damaging attack (or other failure) is recognized, it is necessary to take steps to immediately recover essential services and, eventually, full services. There are a number of techniques that can be used, and many of them have been applied to recovery from failure in the past. Their effects range from transparent maintenance of full services without noticeable interruption to fallback positions that maintain only a core of essential services. The importance of recovery techniques is highlighted when the effects of an attack are considered.

Redundancy is the key to maintaining full services in the face of failures. The fault-tolerance community has considerable experience in the use of redundancy to maintain service in the face of component failures, but their analytical techniques are predicated on knowing the statistical distributions associated with various failure mechanisms, something that may not be possible with failures induced by attacks.

In many cases, the replication of critical data is a primary means for achieving recovery. When essential services are supplied through commodity databases or the like, it may be pos-

sible to restore a critical data service by simply starting up another instance of the commodity server with a replicated database at a more or less arbitrary location.

Systematic backup of all data sources, combined with appropriate mechanisms for restoring the data on either the originating platform or another platform, is a key part of any recovery strategy. The required granularity of backups depends on the frequency at which data changes and the cost of repeating work performed between backups. In extreme cases, it may be necessary to backup files each time they are closed after writing, and to log transactions or keystrokes so that intermediate work can be recovered. In other cases, daily or weekly backups may suffice.

When a system is under attack or has experienced a failure, it may be possible to dynamically reconfigure the system to transfer essential services from the attacked component to an operational one, eliminating less essential services in the process. This strategy is employed by the Federal Reserve, which can tolerate limited outages at one of its three primary computational centers in this fashion. Since this strategy does not have redundant capacity, the reconfiguration can persist only for limited periods, as the criticality of less essential services increases with the length of time that they are unavailable.

Finally, it may be possible to devolve the system to an alternate mode of operation, perhaps one in which the role of the computer system is temporarily reduced or even eliminated. For example, computer-to-computer transactions might be replaced with manually initiated faxes. A computerized parts inventory and order system might revert for a short period to a manual system that indicates reorder levels by red lines on storage bins.

Perhaps the hardest part of survivability is adapting a system to make it more robust in the hope that it will resist never-before-seen attacks or intrusions. Just as attackers are constantly looking for new points of vulnerability, defenders must create defenses that are based on generalizations of previously seen attacks and must try to anticipate the directions from which new attacks might occur.

2.4 Survivability as an Integrated Engineering Framework

As a broadly based engineering paradigm, survivability is a natural framework for integrating established and emerging software engineering disciplines in the service of a common goal. The established areas of software engineering that are related to survivability include security, fault tolerance, safety, reliability, reuse, performance, verification, and testing. Research in survivability encompasses a wide variety of research methods, including the investigation of analogs to the immunological functioning of an individual organism and sociological analogs to public health efforts at the community level.

The discipline of computer security has made valuable contributions to the protection and integrity of information systems over the past three decades. However, "computer security" has traditionally been used as a binary term that suggests that at any moment in time a system is either safe or compromised. We believe that this use of the term engenders viewpoints that largely ignore the aspects of recovery from the compromise of a system, as well as aspects of maintaining services during and after an intrusion. Such an approach is inadequate to support necessary improvements in the state of the practice of protecting computer systems from attack. In contrast, the term *survivable systems* refers to systems whose components collectively accomplish their mission even under attack and despite active intrusions that effectively damage a significant portion of the system.

Robustness under attack is at least as important as hardness or resistance to attack. Hardness contributes to survivability, but robustness under attack (and, in particular, recoverability) is the essential characteristic that distinguishes survivability from traditional computer security. At the same time, survivability can benefit from computer security research and practice, and survivability can provide a framework for integrating security with other disciplines that can contribute to system survivability.

Survivability requires robustness under conditions of intrusion, failure, or accident. The concept of survivability includes fault tolerance, but is not equivalent to it. Fault tolerance relates to the statistical probability of an accidental fault or combination of faults, not to malicious attack. For example, an analysis of a system may determine that the simultaneous occurrence of three statistically independent faults (f1, f2, and f3) will cause the system to fail. The probability of the three independent faults occurring simultaneously by accident may be extremely small, but an intelligent adversary with knowledge of the system's internals can orchestrate the simultaneous occurrence of these three faults and bring down the system. A fault-tolerant system most likely does not address the possibility of the three faults occurring simultaneously, if the probability of occurrence is below a threshold of concern. A survivable system requires a contingency plan to deal with such a possibility.

Redundancy is another factor that can contribute to the survivability of systems. However, redundancy alone is insufficient, because multiple identical backup systems share identical vulnerabilities. A survivable system requires each backup system to offer equivalent functionality, but significant variance in implementation. This variance thwarts attempts to compromise the primary system and all backup systems with a single attack strategy.

3 System Development Life-Cycle Models

3.1 The Spiral Model

Here we describe a life-cycle model that was developed for use in trusted systems [Marmor-Squires 89]. Such a model is a natural fit for development of survivable systems. This work was based on an assessment of the waterfall and spiral models, and an extension of the spiral model to incorporate concepts of trusted systems.

Analysis of life-cycle model work done to date led to widespread use of the TRW spiral model as a foundation. The spiral model can be adapted for use in developing survivable systems. The spiral model for the software development process has been developed at TRW as an alternative to more conventional (largely waterfall-style) models. Its key features are risk management, robustness, and flexibility. This section is devoted to a description of the basic spiral model and a specialization of it. The description of the spiral model largely follows that of Boehm [Boehm 89]. Much of the initial work on spiral models was carried out by Mills and his associates [Mills 86].

The development of software is, at best, a difficult process. Many software systems, especially in the commercial area, simply evolve over time without a well-defined development process. Other systems are developed using (or at least giving lip service to) a stagewise progression of steps, possibly with feedback between adjacent steps, as in the waterfall model [Royce 87]. As Parnas has pointed out, this model makes a much better ex post facto explanation of the development process than a guide for its execution [Parnas 86].

Over the years, numerous variations on, or alternatives to, the waterfall model have been proposed. Each of these overcomes certain defects in the waterfall model, but introduces its own set of additional problems.

While the waterfall model serves a useful purpose in introducing discipline into the software development process, it essentially dictates the linear progressions that were necessary in the batch-oriented world of limited alternatives and scarce computational power. It assumes a factory-like assembly-line system in which each piece is understood. At the present time, the availability of workstations, networks, and inexpensive mass storage, along with a variety of tools, makes possible a wide variety of exploratory programming activities as part of the development process. This means that it is possible to develop prototypes or models for parts of

systems, obtain reactions from a potential user community, and feed this information back into the development process. Standardization efforts have produced large libraries of components and even entire subsystems that can be used to reduce the amount of new development required for a project. The growing availability of development and execution environments has accelerated this trend.

The spiral model is an attempt to provide a disciplined framework for software development that both overcomes deficiencies in the waterfall model and accommodates activities such as prototyping, reuse, and automatic coding as parts of the process. A consequence of the flexibility of the life-cycle model is that the developer is faced with choices at many stages of the process. With choice comes risk; therefore much of the emphasis of the spiral model is placed on risk management. This, in turn, may result in uneven progress in various aspects of system development, with high-risk areas being explored in depth, while low-risk areas are deferred.

The spiral model views the development process in polar coordinates. The r coordinate represents cumulative project cost, the w coordinate represents progress to date. The plane is divided into four quadrants that represent different kinds of activities, as follows:

I. determination of objectives, alternatives, and constraints

II. evaluation of alternatives; identification and resolution of risks

III. development activities

IV. review and planning for future cycles

In addition, the boundary between quadrants I and IV represents a commitment to move forward with a particular element, approach, or method, and advance to the next stage (or spiral) within a defined space of activities (e.g., design). Specific activities may overlap multiple spirals. Also, concurrent spirals may be required to address varying areas of risk. The commitment line may involve a decision to terminate the project or change direction based on the review results.

Figure 1 shows a single cycle of the spiral. The paragraphs that follow characterize the activities that take place in each quadrant. Note that w does not progress evenly with time. Some cycles of the spiral may require months to complete, while others require only days. Similarly, although increasing w denotes progress within a cycle of the spiral, it does not necessarily denote progress towards project completion. Each cycle of the model addresses all the activities between review and commitment events. Early in the process, cycles may be short as alternatives in the decision space of the project are explored. As risks are resolved, cycles may stretch, with the development quadrant subsuming several steps in the waterfall. The spiral may be terminated with product delivery, in which case modification or maintenance activities are new spirals, or the original spiral may continue until the product is retired.

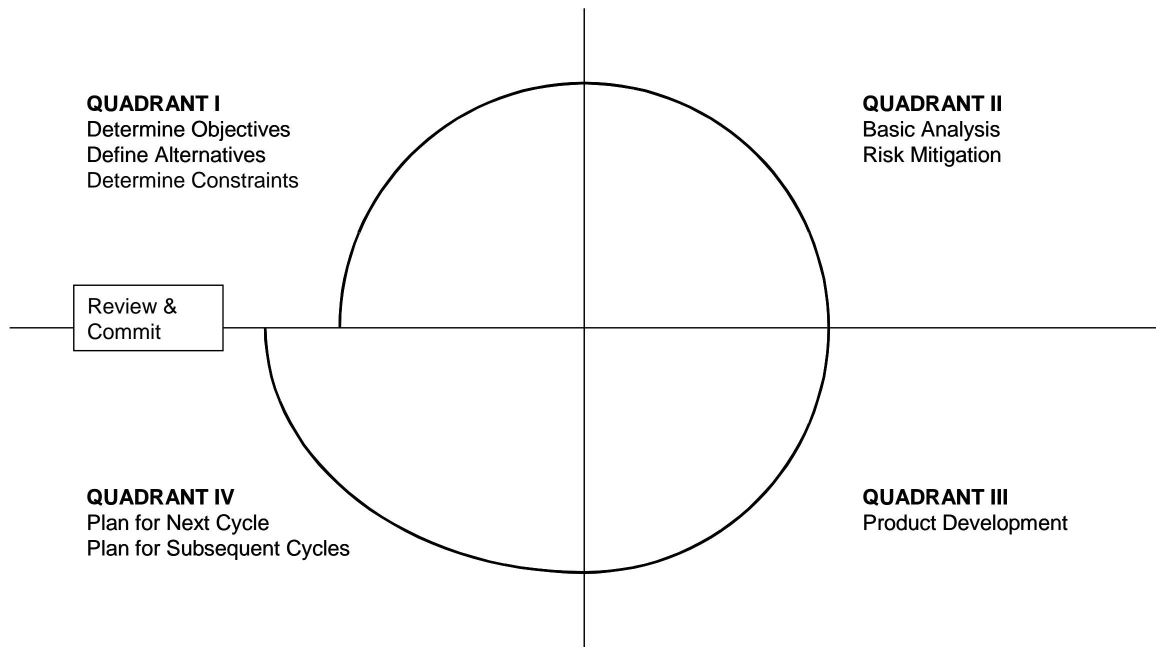

Figure 1: A Project Spiral Cycle

3.2 A Spiral Model for Survivable Systems Development

The generalized "pure" spiral process discussed above provides a framework for more specialized models. Specialization and enhancement call for adapting the activities carried out under the general model to the special requirements of the systems to be produced. This is done by specifying (*a*) activities that address the drivers that characterize the system and (*b*) constraints that characterize the environment in which the system is to be produced.

The primary driver in the present context is the requirement to develop a survivable system. Constraints include the political and social environment in which the system is to be constructed, the ever-present cost considerations, and the limitations of technologies and knowledge that can be brought to bear on the problem at hand. These combine to yield a specialized version of the spiral model that integrates survivability into the management process, as depicted in Figure 2.

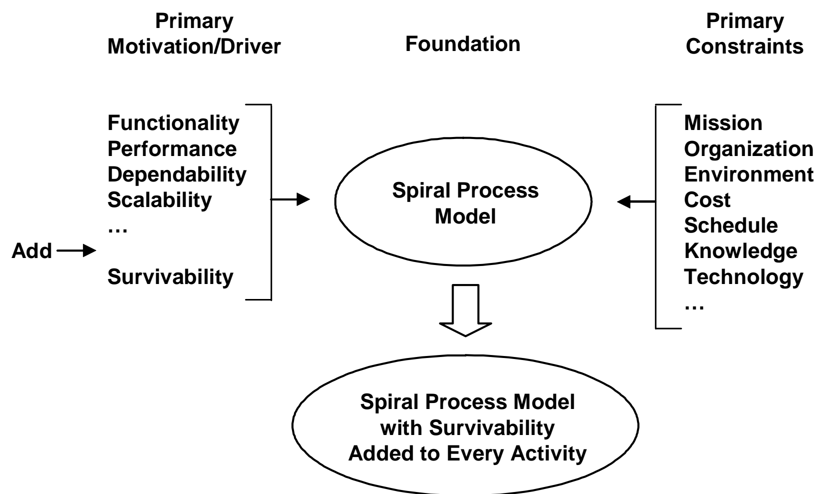

Figure 2: Specialization of the Spiral Model for Survivability Driver

Survivable systems must satisfy a variety of conflicting interests. End users want them to carry out their primary operational mission, possibly at the expense of violating security policies under some circumstances. It is often the case that systems must also satisfy some certification or accreditation authority. The steps required for these approvals may conflict with the interests of users. And developers want to finish the job, preferably ahead of schedule and under budget. Within the development organization, tensions may exist between the various specialties involved. Resolving these conflicts may involve constraining the environment and the development process. In addition, cost considerations are always present. The spiral development process has proven to be more cost effective than traditional methods, but exhibits a different distribution of costs over time. Under the spiral model, expenditures are typically higher in early specification and design activities, resulting in cost savings in later implementation and integration activities.

Table 2 identifies a typical set of broad system-development activities and the corresponding survivability elements of each. The key point is that survivability is integrated into the broader activities. For example, in defining system requirements, function, performance, dependability, scalability, and other properties must be defined, as well as survivability attributes. The activities in Table 2 compose the subject matter for project management under the specialized spiral model of Figure 2.

As an illustration, consider the following imagined application of the spiral management process to the architecture-definition phase. We assume that prior phases have been completed successfully and that the appropriate requirements and specification documents are at hand. The task of the initial architecture-definition spiral is to define a set of candidate com-

ponents and their interconnections that will implement the specified services in a way that satisfies both functional and non-functional requirements. The architect will choose candidate platforms, allocate functions to them, and determine the appropriate connections among platforms and between platforms and the outside world. A variety of tools and techniques will be used to analyze the proposed architecture to determine whether it satisfies the requirements and specifications. One possibility of this analysis is that the proposed architecture satisfies the functional requirements but cannot achieve the required throughput. Although processor replication has already been used to improve performance, the processors require close coupling to maintain synchronization and their co-location presents a vulnerability as a potential single site of failure. Another spiral over the architecture is in order, as unresolved risks remain.

An examination of the specification for the service that results in the bottleneck shows that what appeared as a monolithic service at first glance actually decomposes in a way that reduces the processing load and allows the two parts of the service to be separated both physically and temporally. After confirming that this revised service specification satisfies the requirements and is consistent with the other, unchanged specifications, the architecture is revisited. The revised specification permits a reduction in processor load and allows the critical function to be performed at several distant locations with greatly relaxed data-synchronization requirements. As a result, it is possible to configure the system with sufficient redundancy so that at least two loss-of-site events can be tolerated without loss of service. Further site loss will reduce service levels, but it is possible to prioritize requests so that the minimum essential service level will be maintained. Detailed analyses of this approach show a low probability race condition that could deadlock the system. Adding explicit synchronization mechanisms (another iteration) and additional communications capacity reduces the residual risk to an acceptable level, and the architecture phase is complete after two spirals of the management process.

4 System Development Life-Cycle Activities and Survivability

The key survivability elements of Table 2 are the principal tasks that must be managed within the spiral model to achieve system survivability. In this section, we examine the technologies and processes of several of these elements, including requirements and specification, architecture and design, testing, and evolution.

Table 2: Life-Cycle Activities and Corresponding Survivability Elements

Life-Cycle Activities	Key Survivability Elements	Examples
Mission definition	Analysis of mission criticality and consequences of failure	Estimation of cost impact of denial-of-service attacks
Concept of operations	Definition of system capabilities in adverse environments	Enumeration of critical mission functions that must withstand attacks
Project planning	Integration of survivability into life-cycle activities	Identification of defensive coding techniques for implementation
Requirements definition	Definition of survivability requirements from mission perspective	Definition of access requirements for critical system assets during attacks
System specification	Specification of essential service and intrusion scenarios	Definition of steps that compose critical system transactions
System architecture	Integration of survivability strategies into architecture definition	Creation of network facilities for replication of critical data assets
System design	Development and verification of survivability strategies	Verification of data-encryption algorithms for correctness
System implementation	Application of survivability coding and implementation techniques	Definition of methods to avoid buffer overflow vulnerabilities
System testing	Treatment of intruders as users in testing and certification	Addition of intrusion usage to usage models for statistical testing; use of independent verification and validation
System evolution	Improvement of survivability to prevent degradation over time	Redefinition of architecture in response to changing threat environment

4.1 Requirements and Specification

Requirements elicitation, validation, and specification are key early steps in the system life cycle. Survivability requirements can vary substantially depending on system scope, criticality, and the consequences of failure and interruption of service. Categories of requirements definition for survivable systems include function, usage, development, operation, and evolu-

tion. In this section, we discuss the nature of survivability requirements, how these requirements can be expressed, and their impact on system survivability.

For typical large-scale systems, the new paradigm for requirements definition is characterized by distributed hardware, services, and code (including executable content), distributed and shared communications and routing infrastructure, diminished trust, and a lack of unified administrative control. Assuring survivability of mission-critical systems developed under this new paradigm is a formidable high-stakes effort for software engineering research. This effort requires that traditional computer security measures be augmented by new and comprehensive system survivability strategies.

4.1.1 Expressing Survivability Requirements

The definition and analysis of survivability requirements is a critical first step in achieving system survivability [Linger 98]. Figure 3 depicts an iterative model for defining these requirements. Survivability must address not only requirements for software functionality, but also requirements for software usage, development, operation, and evolution. Thus, five types of requirements definitions are relevant to survivable systems in the model. These requirements are discussed in detail in the subsections that follow.

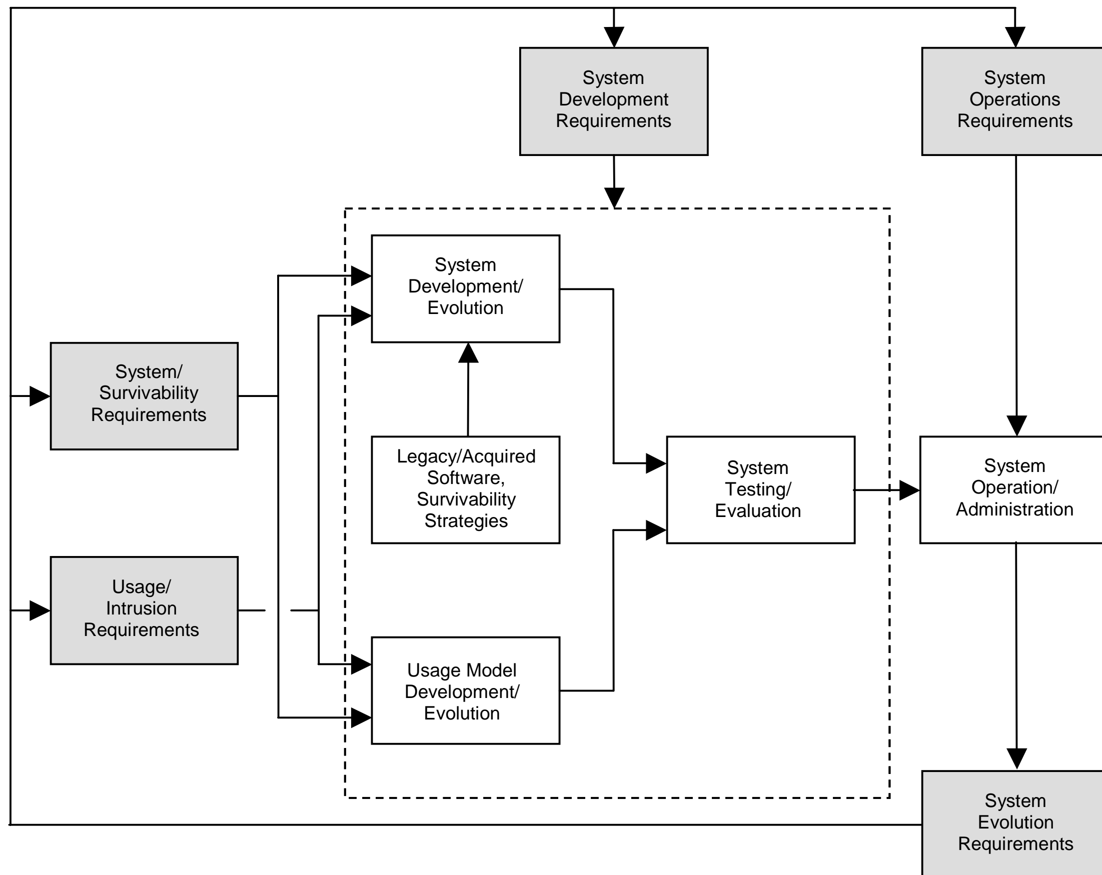

Figure 3: Classes of Requirements for Survivable Systems

System/Survivability Requirements. The term *system requirements* refers to traditional user functions that a system must provide. For example, a network management system must provide functions to enable users to perform such tasks as monitoring network operations and adjusting performance parameters. System requirements also include non-functional aspects of a system, such as timing, performance, and reliability. The term *survivability requirements* refers to the capabilities of a system to deliver essential services in the presence of intrusions and compromises and to recover full services.

Figure 4 depicts the integration of survivability requirements with system requirements at node and network levels.

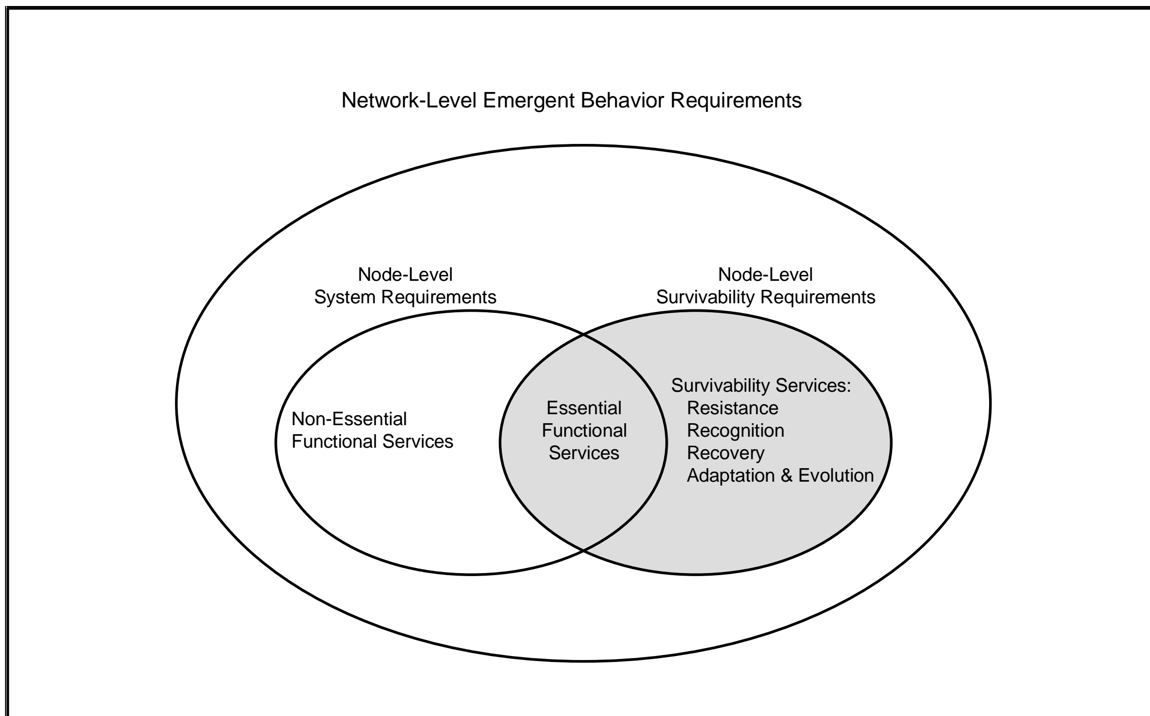

Figure 4: Integrating Survivability into System Requirements

Survivability requires that system requirements be organized into essential services and non-essential services. Essential services must be maintained even during successful intrusions; non-essential services are recovered after intrusions have been handled. Essential services may be stratified into any number of levels, each embodying fewer and more vital services as the severity and duration of intrusion increases. Thus, definitions of requirements for essential services must be augmented with appropriate survivability requirements.

As shown in Figure 3, survivable systems may also include legacy and acquired COTS components that were not developed with survivability as an explicit objective. Such components may provide both essential and non-essential services and may require functional requirements for isolation and control through wrappers and filters to permit their safe use in a survivable system environment.

Figure 4 shows that survivability itself imposes new types of requirements on systems. These new requirements include the resistance to, recognition of, and recovery from intrusions and compromises, and adaptation and evolution to diminish the effectiveness of future intrusion attempts. These survivability requirements are supported by a variety of existing and emerging *survivability strategies*, as noted in Linger et al.'s paper [Linger 98], and are discussed in more detail below.

Finally, Figure 4 depicts *emergent behavior requirements* at the network level. These requirements are characterized as *emergent* because they are not associated with particular

nodes, but rather emerge from the collective behavior of node services in communicating across the network. These requirements deal with the survivability of overall network capabilities (e.g., capabilities to route messages between critical sets of nodes regardless of how intrusions may damage or compromise network topology).

We envision survivable systems that are capable of adapting their behavior, function, and resource allocation in response to intrusions. For example, when necessary, functions and resources devoted to non-essential services could be reallocated to the delivery of essential services and to intrusion resistance, recognition, and recovery. Requirements for such systems must also specify how the system should adapt and reconfigure itself in response to intrusions.

Systems can exhibit large variations in survivability requirements. Small local networks may require few or no essential services and recovery times may be measured in hours. Conversely, large-scale networks of networks may require a core set of essential services, automated intrusion detection, and recovery times measured in minutes. Embedded command-and-control systems may require essential services to be maintained in real time with recovery times measured in milliseconds.

The attainment and maintenance of survivability consumes resources in system development, operation, and evolution. The resources allocated to a system's survivability should be based on the costs and risks to an organization associated with the loss of essential services.

Usage/Intrusion Requirements. Survivable-system testing must demonstrate the correct performance of essential and non-essential system services as well as the survivability of essential services under intrusion. Because system performance in testing (and operation) depends totally on the system's use, an effective approach to survivable-system testing is based on usage scenarios derived from usage models [Mills 92, Linger 99b].

Usage models are developed from usage requirements. These requirements specify usage environments and scenarios of system use. Usage requirements for essential and non-essential services must be defined in parallel with system and survivability requirements. Furthermore, intruders and legitimate users must be considered equally. Intrusion requirements that specify intrusion-usage environments and scenarios of intrusion use must be defined as well. In this approach, intrusion use and legitimate use of system services are modeled together.

Intruders might engage in scenarios beyond legitimate scenarios, but they might also employ legitimate use for purposes of intrusion if they gain the necessary privileges.

Development Requirements. Survivability places stringent requirements on system development and testing practices. Inadequate functionality and software errors can have a devas-

tating effect on system survivability and provide opportunities for intruder exploitation. Sound engineering practices are required to create survivable software.

The following five principles (four technical and one organizational) are example requirements for survivable-system development and testing practices:

- Precisely specify required functions of a system in all possible circumstances of use.
- Verify the correctness of system implementations with respect to functional specifications.
- Precisely specify function usage in all possible circumstances of system use, including intruder use.
- Test and certify the system based on function usage and statistical methods.
- Establish permanent readiness teams for system monitoring, adaptation, and evolution.

Sound engineering practices are required to deal with legacy and COTS software components as well.

Operations Requirements. Survivability places demands on requirements for system operation and administration. These requirements include defining and communicating survivability policies, monitoring system use, responding to intrusions, and evolving system functions as needed to ensure survivability as usage environments and intrusion patterns change over time.

Evolution Requirements. Systems evolution responds to user requirements for new functions. However, this evolution is also necessary to respond to increasing intruder knowledge of system behavior and structure. In particular, survivability requires that system capabilities evolve more rapidly than intruder knowledge. This rapid evolution prevents intruders from accumulating information about otherwise invariant system behavior, which intruders need to achieve successful penetration and exploitation.

4.1.2 Requirements Definition for Essential Services

The preceding discussion distinguishes between essential and non-essential services. Each system requirement must be examined to determine whether it corresponds to an essential service. The set of essential services must form a viable subsystem for users that is complete and coherent. If multiple levels of essential services are required, each set of services provided at each level must also be examined for completeness and coherence. In addition, requirements must be defined for making the transition to and from essential-service levels.

When distinguishing between essential and non-essential services, all of the usual requirements-definition processes and methods can be applied. Elicitation techniques such as those

embodied in software requirements engineering can help to identify essential services [Ebert 97]. Tradeoff and cost/benefit analysis can help to determine the sets of services that sufficiently address business survivability risks and vulnerabilities. Provisions for tracing survivability requirements through design, code, and test must be established. As previously mentioned, simulations of intrusion through intruder-usage scenarios are included in the testing process.

4.1.3 Requirements Definition for Survivability Services

After requirements are specified for essential and non-essential services, a set of requirements for survivability services must be defined. These services can be organized into four general categories: resistance, recognition, recovery, and adaptation and evolution. These survivability services must operate in an intruder environment that can be characterized by three distinct phases of intrusion: penetration, exploration, and exploitation.

Penetration Phase. In this phase, an intruder attempts to gain access to a system through various attack scenarios. These scenarios range from random inputs by hobbyist hackers to well-planned attacks by professional intruders. These attempts are designed to capitalize on known system vulnerabilities.

Exploration Phase. In this phase, the system has been penetrated and the intruder is exploring internal system organization and capabilities. By exploring, the intruder learns how to exploit the access to achieve intrusion objectives.

Exploitation Phase. In this phase, the intruder has gained access to desired system facilities and is performing operations designed to compromise system capabilities.

Penetration, exploration, and exploitation create a spiral of increasing intruder authority and a widening circle of compromise. For example, penetration at the user level is typically a means to find root-level vulnerabilities. User-level authorization is then employed to exploit those vulnerabilities to achieve root-level penetration. Finally, compromise of the weakest host in a networked system allows that host to be used as a stepping-stone to compromise other more protected hosts.

Requirements definitions for resistance, recognition, recovery, and adaptation and evolution services help developers select survivability strategies to deal with these phases of intrusion. Some strategies, such as firewalls, are the product of extensive research and development and currently are used extensively in bounded networks. New survivability strategies are emerging to respond to the unique challenges of unbounded networks.

Resistance Service Requirements. *Resistance* is the capability of a system to deter attacks. Resistance is thus important in the penetration and exploration phases of an attack, before actual exploitation. Current strategies for resistance include the use of firewalls, authentication, and encryption. Diversification is a resistance strategy that will likely become more important for unbounded networks.

Requirements for diversification must define planned variations in a survivable system's function, structure, and organization, and the means for achieving those variations. Diversification is intended to create a *moving target* and render ineffective the accumulation of system knowledge as an intrusion strategy. Diversification also eliminates intrusion opportunities associated with multiple nodes that execute identical software and typically exhibit identical vulnerabilities. Such systems offer tempting economies of scale to intruders, because when one node has been penetrated, all nodes can be penetrated. Requirements for diversification can include variation in programs, retained data, and network routing and communication. For example, systematic means can be defined to randomize software programs while preserving functionality [Linger 99a].

Recognition Service Requirements. Recognition is the capability of a system to recognize attacks or the probing that precedes attacks. The ability to react or adapt during an intrusion is central to a system's capacity to survive an attack that cannot be completely repelled. To react or adapt, the system must first recognize it is being attacked. In fact, recognition is essential in all three phases of attack.

Current strategies for attack recognition include both state-of-the-art intrusion detection and mundane but effective techniques such as logging and frequent auditing, as well as follow-up investigations of reports generated by ordinary error-detection mechanisms. Advanced intrusion-detection techniques are generally of two types: anomaly detection and pattern recognition. *Anomaly detection* is based on models of normal user behavior. These models are often established through statistical analysis of usage patterns. Deviations from normal usage patterns are flagged as suspicious. *Pattern recognition* is based upon models of intruder behavior. User activity that matches a known pattern of intruder behavior raises an alarm.

Requirements for future survivable networks will likely employ additional strategies such as self-awareness, trust maintenance, and black-box reporting. *Self-awareness* is the process of establishing a high-level semantic model of the computations that a component or system is executing or has been asked to execute. A system or component that *understands* what it is being asked can refuse requests that would be dangerous, would compromise a security policy, or would adversely affect the delivery of minimum essential services.

Trust maintenance is achieved by a system through periodic queries among its components (e.g., among the nodes in a network) to continually test and validate trust relationships. Detection of signs of intrusion would trigger an immediate test of trust relationships.

Black-box reporting is a dump of system information that can be retrieved from a crashed system or component for analysis to determine the cause of the crash (e.g., a design error or a specific intrusion type). This analysis can help to prevent other components from suffering the same fate.

A survivable-system design must include explicit requirements for recognition of attack. These requirements ensure the use of one or more of the aforementioned strategies through the specification of architectural features, automated tools, and manual processes. Because intruder techniques are constantly advancing, recognition requirements should be frequently reviewed and continuously improved.

Recovery Service Requirements. *Recovery* is a system's ability to restore services after an intrusion has occurred. Recovery also contributes to a system's ability to maintain essential services during intrusion.

Requirements for recoverability are what most clearly distinguish survivable systems from systems that are merely secure. Traditional computer security leads to the design of systems that rely almost entirely on hardening (i.e., resistance) for protection. Once security is breached, damage may follow with little to stand in the way. The ability of a system to react during an active intrusion is central to its capacity to survive an attack that cannot be completely repelled. Recovery is thus crucial during the exploration and exploitation phases of intrusion.

Recovery strategies in use today include replication of critical information and services, use of fault-tolerant designs, and incorporation of backup systems for hardware and software. These backup systems maintain master copies of critical software in isolation from the network. Some systems, such as large-scale transaction processing systems, employ elaborate, fine-grained transaction roll-back processes to maintain the consistency and integrity of state data.

Adaptation and Evolution Service Requirements. Adaptation and evolution are critical to maintaining resistance to ever-increasing intruder knowledge of how to exploit otherwise unchanging system functions. Dynamic adaptation permanently improves a system's ability to resist, recognize, and recover from intrusion attempts. For example, an adaptation requirement may be an infrastructure that enables the system to inoculate itself against newly discovered security vulnerabilities by automatically distributing and applying security fixes to all network elements. Another adaptation requirement may be that intrusion-detection rule sets are updated regularly in response to reports of known intruder activity from authoritative sources of security information, such as the CERT Coordination Center.

Adaptation requirements ensure that such capabilities are an integral part of a system's design. As in the cases of resistance, recognition, and recovery requirements, the constant evolution of intruder techniques requires that adaptation requirements be frequently reviewed and continuously improved.

4.2 Architecture and Design

The architectural level of a Survivable Network Design (SND) method is depicted in Figure 5. In contrast to treating survivability as an after-the-fact add-on to system architecture and design, SND integrates survivability considerations into the development process. The SND method is based on the Survivable Systems Analysis process [Ellison 98, Ellison 99b, Mead 00a] developed and applied by the CERT Coordination Center.

SND is driven by several of the requirements-specification types depicted in Figure 3, specifically

- system/survivability requirements. System requirements define all possible functional behavior, plus non-functional properties that a system must satisfy. Survivability requirements identify those elements of functional behavior that represent essential services, and elaborate these services in terms of essential service scenarios of use.
- usage/intrusion requirements. These requirements define all possible system usage under normal and adverse circumstances. Intruders are treated as another class of users, and representative intrusion scenarios of use are defined.
- operations/administration requirements. These requirements identify operational procedures and policies (security policies, for example) that must be developed.

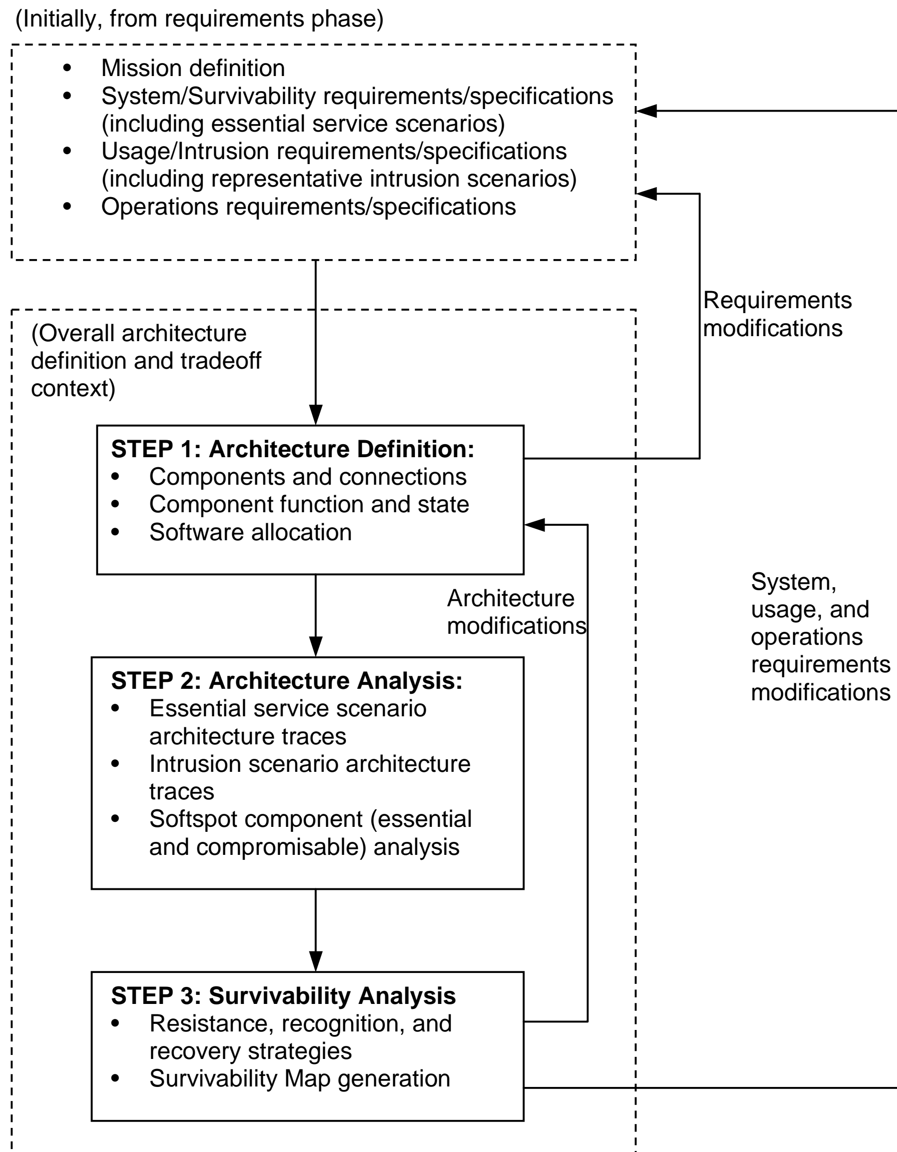

Figure 5: Architectural Level of a Survivable Network Design Method

The architectural application of the SND method is embedded within a broader activity of architecture definition and tradeoff that seeks to create a system architecture that satisfies all required properties, such as performance, capacity, scalability, cost, and maintainability. The focus of Figure 5 is on the SND method, in the knowledge that, in practice, this survivability-specific process will be performed in parallel with other forms of analysis and design, including, for example, simulation and rate monotonic analysis to predict performance properties. The SEI's Architecture Tradeoff Analysis Method[SM] (ATAM[SM]) [Kazman 98] is directed at

[SM] Architecture Tradeoff Analysis Method and ATAM are service marks of Carnegie Mellon University.

integrating various forms of analysis and design to create an architecture that best satisfies possibly conflicting requirements. The architectural level of SND is composed of three steps, as follows:

Step 1. Architecture Definition: In this step, a proposed architecture is developed based on the system mission and requirements. Architectural styles and patterns are selected, and components and connectors are defined. Components can be described in terms of functionality to be provided and state data to be retained. Typically, specific software elements are identified, including protocols, operating systems, execution environments, and applications.

Step 2. Architecture Analysis: This step investigates the survivability properties of a candidate architecture in terms of (1) essential services (services that must be maintained during attack); (2) essential assets (assets whose integrity, confidentiality, availability, and other properties must be maintained during attack); and (3) mission objectives and the consequences of failure. First, essential service and asset uses are characterized by usage scenarios that are mapped onto the architecture as execution traces to identify the set of corresponding *essential components* (components that must be available to deliver essential services and maintain essential assets). Next, representative intrusion scenarios are selected based on the system environment and an assessment of risks and intruder capabilities. Selections are also influenced by the extensive CERT knowledge base of intrusion strategies. These scenarios are likewise mapped onto the architecture as execution traces to identify corresponding sets of *compromisable components* (components that could be penetrated and damaged by intrusion). In this mapping, the SND method takes into account strengths and weaknesses of COTS components, as well as any known security and reliability flaws. Finally, *softspot components* of the architecture are identified as components that are both essential and compromisable, based on the previous results.

Step 3: Survivability Analysis: Softspot components and their supporting architectures are then analyzed for the key survivability properties of resistance, recognition, and recovery. At this point, survivability strategies and architectural patterns are evaluated for potential use in improving the survivability of the candidate architecture. This analysis of the “three Rs” is summarized in a *Survivability Map.* The map is a matrix that enumerates, for every intrusion scenario and its corresponding softspot effects, the current and recommended *architecture strategies* for resistance, recognition, and recovery. The Survivability Map provides feedback to the original architecture and often results in an iterative process of cost/benefit analysis and survivability improvement. It can also provide feedback to system requirements, as new understandings and better ideas emerge from the analysis.

The scenario-based approach in SND is a generalization of operation-sequence [Kemmerer 91] and usage-scenario methods [Carrol 99, Prowell 99].

4.3 Implementation and Verification

4.3.1 Defensive Coding Strategies

Many intrusion vulnerabilities turn out to be the result of poor coding practices that result in unintended but exploitable program behavior that intruders can employ to gain access. Management of survivable systems development thus requires that defensive coding standards and practices be defined and enforced.

System specifications typically deal with high-level behaviors under the assumption that programs operate within the constraints imposed by the semantics of both the specification language and the programming or implementation language. In general, specifications assume that operation in the mathematical world of real numbers and arbitrarily large integers. However, programming languages operate with floating-point numbers and integers with a restricted range. Because we would like to show that program behavior is not inconsistent with specified behavior, special attention is required to ensure that program operations produce values that remain within these ranges and, in the integer case at least, are identical to the mathematically expected results.

Programming languages vary in their treatment of programs that violate these constraints. In general, we are concerned with a class of constraints that specifies the legitimate values that a variable of a given data type may assume. These, in turn, limit the operations that can be performed on such variables. For example, the result of performing an integer division with a divisor of zero is undefined and the result of such a division cannot, in general, be assigned to an integer variable. Similarly, the results of arithmetic operations that result in hardware overflows are not consistent with the mathematical definitions of the same operations.

Languages such as Ada provide exception mechanisms that detect attempts to violate language assumptions at run time. Programmers can choose to provide code to deal with these exceptional conditions and attempt recovery or they can allow the run-time system to terminate the program when a constraint is violated. Languages such as C leave the behavior of exceptional programs undefined (or defined by the implementation) and provide little or no help to the programmer in either avoiding exceptions or coping with them when they occur. The effect of an overflow in a C program is typically to assign a legal, but incorrect, result to a variable. Subsequent use of such a result may lead to a cascade of wrong results that culminates in either erroneous program results being given to a user or in a problem so egregious that the program abnormally terminates, perhaps from a hardware addressing error.

When an array is accessed, the subscript used must reference an element within the declared bounds of the array. Similarly, when whole arrays are manipulated (e.g., strings are concatenated), it is assumed that the target of the operation is large enough to hold the entire result. Since arrays are typically mapped onto contiguous blocks of address space, reading from an

address that is not part of the array has one of two effects: (1) the address is not a part of the program's address space at all and some sort of addressing fault occurs, or (2) the address is allocated to something else—code, another variable, or unused, but addressable memory (slack space). In the first case, an abnormal program termination usually results. In the second case, the results are unpredictable unless the precise mapping of code, data, and slack space is known. If the array operation is a store, the result could be an unexpected change in the value of another variable or the corruption of program code. If one knows how code and data are allocated by the compiler and loader, it is possible to use out-of-bounds array operations to modify code and achieve predictable changes in program behavior. Starting with the Morris worm of 1988, this has become a common method for attacking Internet programs.

For the most part, both scalar- and array-type violations are almost completely avoidable. Languages with suitable exception semantics, such as Ada, provide run-time indications of attempts to access out-of-bounds array elements. Type-safe languages, such as Java, provide similar protections. In C, programmers can code explicit checks to overflows and out-of-bounds accesses. This is not often done for several reasons. It is commonly thought that such checks exact an unacceptable run-time cost. In addition, most programmers simply do not think about the ways in which their program could be forced to fail by supplying abnormal inputs. Defensive programming techniques can prevent a broad class of program failures that result from abnormal inputs. If they are carefully applied, these techniques need not exact a significant run-time penalty. In general, the program specification should describe the input ranges over which the program is expected to work.

The first step in constructing a defensive system is to add code to check that the data is within the expected range. In the case of arrays or structures, this means reading data in bounded chunks. The original Morris worm took advantage of the fact that the fingered program assumed that its input lines would be the correct size (92 bytes) and used a library routine (gets) that read until a "newline" character was encountered, placing the characters read into an array. A defensive version of this would count characters as it read, discarding inputs that were either too short or too long and then making sure that inputs of the correct length had the appropriate structure.

Once the inputs have been validated, it is usually possible to reason about additional exceptions. If the lengths of strings being concatenated have been previously checked, they can be safely combined if the target of the concatenation is large enough to hold their combined maximum lengths. Similar reasoning applies to exceptions that might arise from arithmetic operations. Experience with a variety of programs shows that, aside from the input checks (which should always be performed since the inputs are not under the programmer's control) few, if any, additional checks are typically required. Optimizing compilers for type-safe languages typically perform such reasoning in order to eliminate run-time checks [McHugh 84]. These and related techniques are required for systems intended to be evaluated at the higher

assurance levels of the Trusted Computer Security Evaluation Criteria [DoD 85], and Young and McHugh discuss them in detail [Young 87].

Design and code inspections are an excellent method for ensuring uniform application of defensive practices. Inspections can be integrated into the spiral management process as a means to reveal and improve development performance and adherence to project standards.

4.3.2 Correctness Verification

Most intrusion vulnerabilities are the result of poor programming practices that produce unforeseen software behavior which is often useful for intrusion purposes—for example, the unintended and widely exploited behavior associated with buffer-overflow problems. The first and best line of defense against intrusion is software whose required behavior in all possible circumstances of use is fully specified, and whose implementation behavior has been verified against those specifications.

Because systems that can be attacked at all levels must be defended at all levels, it is important to verify all software components for correctness with respect to specifications. By its very nature, testing is insufficient for this purpose. Even small software systems exhibit a virtually infinite population of possible executions; even the most carefully conceived program of testing can exercise no more than a minute fraction of these executions. All testing is really sampling from an essentially infinite population of possible executions. Correctness verification, on the other hand, is intended to examine the full functional behavior of software, and is not limited to particular execution paths.

Function-theoretic verification is particularly well suited for this purpose [Linger 99b, Prowell 99]. This approach permits development teams to completely verify the correctness of software with respect to specifications. A *correctness theorem* defines conditions to be met for achieving correct software. These conditions are verified through systematic and repeatable correctness reasoning patterns applied in special team reviews. While programs contain an essentially infinite number of paths, they are composed of a finite number of nested and sequenced control structures (sequence, ifthenelse, whiledo, etc.). The Correctness Theorem is based on verifying control structures, a finite task, and not on tracing execution paths, an open-ended task. This reduction of verification to a finite process permits all software logic to be checked for correctness, to help ensure that unforeseen behavior and potential intrusion vulnerabilities are eliminated from designs.

The correctness conditions defined by the Correctness Theorem for the fundamental control structures are given in Table 3. The control structures are expressed in generic design language format. On the left, each control structure is preceded by a function definition labeled f that defines and documents its net effect on data—that is, the specified mapping from domain to range that the control structure is to implement. Within the structures, g and h represent

operations on data. On the right, the sequence-correctness question involves function composition, the ifthenelse, case analysis, and the whiledo, a termination argument plus case analysis and function composition combined. Prowell and Stavely provide a full explanation of function-theoretic verification [Prowell 99, Stavely 99].

Table 3: Correctness Conditions for Functional Verification

Control Structure	**Correctness Question**
Sequence: [f] do g; h enddo	Does g followed by h do f?

Ifthenelse:
 	
[f]
if
p
then
g
else
h
endif
	When p is true,
	does g do f,
	and when p is false,
	does h do f?

Whiledo: [f] while p do g enddo	Does the loop terminate, and when p is true, does g followed by f do f, and when p is false, does doing nothing do f?

4.4 Testing

In managing the development of survivable systems, it is important to treat survivability testing on a par with testing for functionality, performance, and other system attributes. Penetration testing and statistical, usage-based testing are two useful approaches for evaluating system survivability.

4.4.1 Penetration Testing

Often called "red teams," groups that engage in penetration testing attempt to compromise a system, in a benign manner, to assess the effectiveness of the system's defenses against cyber-attack. Penetration testing offers a complementary method of assessing the security of a system, but it is never a substitute for traditional system testing or certification. Penetration testing is carried out with the permission of the organization that owns the system, and within the bounds of ground rules specifying what is off limits and what is not. For maximum effectiveness, the test team should be free to use a wide variety of information-gathering tech-

niques, including scanning tools, social engineering, and dumpster diving, to support its subsequent benign attacks on the system. This allows the team to test the security of the organization as a whole, of which the system is only a part.

Incorporating the concept of survivability into the penetration testing approach can greatly increase its effectiveness and value. Computer security concepts give the test team little guidance as to what within a system is worthy of attack. Survivability gives much stronger guidance, because only if mission-critical services have been interrupted should the efforts of the test team be regarded as successful. Successfully attacking non-essential parts of a system does not allow the penetration test team to declare victory. This "strategic" use of penetration testing can be tied to a system's life cycle and to the evolutionary design of survivable systems. As a system's mission is modified (i.e., evolves over time), the essential services that support the mission may change, thus varying the targets that a penetration test team must successfully attack to overcome the system's survivability strategies.

4.4.2 Statistical Usage-Based Testing

As noted earlier, any process of testing can execute only a small sample of possible system executions. The problem and opportunity in testing is how to draw the sample. It turns out that if the sample is representative of eventual field usage, testing results can provide scientifically valid predictions of field experience with the software. In this approach, testing is conducted as a statistical experiment, and the results can be used to predict in statistical terms how the software will behave for all the executions not tested. This statistical, usage-based approach to testing permits certification of software's fitness for use, and is fully described in Prowell et al.'s book [Prowell 99]. In general terms, the process begins by constructing a usage model that enumerates possible software uses and their probabilities of occurrence. Usage models are often expressed in terms of formal grammars or Markov chains. The model can then be sampled according to the probabilities, to identify a set of test cases that is faithful to the defined probability distribution. These cases can be executed, and their outcomes (success or failure) used to predict eventual field experience with the software.

In the context of testing for survivability, intruders can be treated as simply another class of system users. Intruder usage can be integrated into a usage model along with legitimate usage. When the model is sampled, intrusion usage will appear in the test cases together with legitimate usage. Success or failure of intrusion uses can be used to evaluate and predict survivability properties in field use.

4.5 System Evolution

Evolutionary design is an important concept that permeates the life cycle of all complex information systems, but evolution plays a particularly crucial role in the life cycle of surviv-

able systems. That is because the primary focus of information survivability is on protecting a system's mission from the malicious actions of intelligent adversaries. The capabilities of intelligent adversaries are not static, but evolve over time in strength and pervasiveness. The sophistication of attack techniques is constantly evolving. Both awareness of these techniques and automation support in the form of readily available attack scripts and toolkits are continually diffusing throughout the Internet. Moreover, the CERT Coordination Center and other incident-response teams are seeing evidence of an improvement in the software engineering techniques employed in the design of some recent attack scripts. All this translates into an ever-escalating arms race between attackers and defenders that will continue as long as networked software systems exist.

Survivability is fundamentally a discipline that blends computer security with business risk management [Lipson 99]. Perpetual system evolution, based on continual risk assessment over the course of the system life cycle, is central to the design of survivable systems. In a typical maintenance environment, the original architectural vision is not preserved and the integrity of the system degrades over time. In the absence of the perpetual, risk-managed evolution of a system's design, the security and survivability of the system will also degrade over time. For example, new vulnerabilities in many systems' underlying COTS components are continually being discovered, and system configurations drift from their optimal settings. Mission and survivability requirements can change and may no longer be reflected in the design of the existing system. Finally, as stated above, attack techniques are continually evolving and may exceed a system's capacity for automatic adaptation.

We distinguish the evolutionary design of survivable systems from straightforward (possibly automated) adaptation and maintenance activities, such as updating virus definitions, adding new rules and attack patterns to a system's intrusion-detection facility, tuning a firewall, or monitoring security advisories and patching announced security vulnerabilities in COTS components. On the other hand, more complex maintenance activities, which may include new or enhanced capabilities, would be considered part of evolutionary design. The successful evolutionary design of survivable systems is dependent upon the establishment of a "survivability watch" activity, which involves the continual monitoring of the system and its environment to detect changes that may affect the risk-management assumptions upon which the survivability of the system is founded. This argues strongly for the formation of a survivability risk-assessment team (SRT), which would be responsible for the survivability watch activity within the system design team. The resources devoted to the SRT and survivability watch will depend upon executive management's risk tolerance and their perception of the cost–benefit ratio for this activity.

Risk assessments for survivability require a broad range of perspectives and skills, and so the members of the SRT must be drawn from all levels of an organization (executive management, application domain experts, computer security experts, and other stakeholders, including customers). SRTs for particular industry or government sectors can be formed to provide

some generic assistance for organizational SRTs, but the mission-sensitive nature of survivability means that SRTs at the organizational level must bear the ultimate responsibility for survivability risk assessment.

We use the term “risk-assessment triggers” to refer to the elements of a system or its environment that an SRT should monitor, looking for changes that can affect the risk-management assumptions that underlie a system’s survivability. It is up to the SRT to determine if a particular change or set of changes will trigger an evolutionary design activity and to decide upon the extent of that activity. Table 4 contains a representative set of risk-assessment trigger elements that an SRT might track for changes. Trigger events include changes in attack techniques, mission, management, staff, customers, and technological and legal environments.

Table 4: Trigger Elements for Evolutionary-Design Activities for Survivable Systems

Trigger Elements for Evolutionary Design Activity	**Examples**
Attack techniques	A new attack technique or variation has been discovered for which the system cannot adapt automatically or cannot be protected through routine maintenance (e.g., simply by adding a new rule for resistance, recognition, or recovery).
Mission, essential services, essential quality attributes, key information resources and assets	The organization’s mission has changed or the system will be purchased and deployed by other organizations with different missions.
Customers	New customers may be less known (hence less trustworthy), may require more extensive access to information resources and assets, or may require a higher quality of service (e.g., increased availability) than previous customers required.
Management	New executive management may differ in its tolerance for risk and its risk-management strategies.
Workflow and processes	Changes in organizational processes to which the system contributes may affect the overall survivability of the mission. There may be new ways to attack the system or human-machine interface.
Organizational staff	Turnover may result in reduction of staff expertise, which may stress the survivability of the system. In a rapidly growing organization, new staff may be less worthy of trust than previous staff (e.g., there may be less time for background checks or employees may be stationed more remotely).
Vendors	A new vendor for a system component may require remote maintenance and trusted access.
Collaborators	A partner on one project may be a competitor on the next.
Dependencies and interdependencies	Increases in dependency upon a system may be brought on by the elimination of manual processes, staff positions, or legacy systems, which means there is no longer an alternative if the system fails. Another example is the steadily increasing interdependency among the nation’s critical infrastructures.

Trigger Elements for Evolutionary Design Activity	Examples
Technology	A change in the technological environment in which a system operates can reduce the effectiveness of a given security or survivability strategy. (This includes changes in the unbounded systems environment, new security techniques, and changes in the availability and knowledge of technology in the application domain.)
Threat environment	More aggressive industrial competitors or increases in nation-state sponsored cyber-terrorism may require additional system resources to be devoted to survivability.
System components	A COTS component that is no longer supported may have been replaced with a new component whose contribution, positive or negative, to system survivability must be evaluated.
Usage, functionality, access, or quality of service	New means of access to a system (e.g., wireless), new ways of using an existing system, or new expectations for quality of service can affect a system's survivability.
Cost, profit, and other economic factors	Changing cost factors may threaten or improve a system's survivability by affecting the cost–benefit ratio associated with various survivability solutions (e.g., risk mitigation strategies). Affordability is a primary factor that is traded off against survivability. Lowered component cost can lead to an evolutionary redesign providing increased redundancy and diversity to support greater survivability. Increased stockholder demands for short-term profits may tilt survivability requirements toward higher risk.
Legal environment	Use of a system in a new and stricter jurisdiction may increase the risks of liability and threaten survivability. New laws, increased enforcement of existing laws, or lawsuits can change the risk equation and threaten the mission.
Government regulation	Changes in government regulations to support increased privacy, safety, competition, or quality of service may trigger the need to modify a system's design to ensure its continued survivability. (For example, deregulation of the U.S. electric power system increases competition, but may decrease reliability.)
Certification requirements or standards	Business interruption insurance that covers cyber-attacks may depend on certification of the survivability of a system or on demonstration that it meets a minimum standard. Thus, new or changed standards or certification requirements may affect survivability.
Political and social environment	Changes in privacy concerns, trust relationships, or the overall risk tolerance of society will affect the survivability requirements of critical national infrastructures upon which society depends.
Operational experience (attacks, accidents, and failures)	Feedback from field use may lead to the discovery of new threats to a system's survivability or may reveal deficiencies.
Results of periodic penetration testing and survivability evaluations (SSAs)	Troublesome results from regularly scheduled red team penetration testing and security/survivability evaluations can trigger awareness of the need for evolutionary improvements.

A change in one or more of the trigger elements can initiate any of a broad range of evolutionary design activities described in Table 5, ranging from no action at all, to performing one or more survivability life-cycle activities, to abandonment of a system. The SRT would initiate the design activities, but the system design team would be responsible for performing them.

Table 5: Possible Evolutionary-Design Activities in Response to a Trigger Event

Evolutionary Design Activity	**Example**
No action needed or taken.	Executive management sees no new threat to a system's mission posed by greatly increased hiring activity, because all new hires are subject to thorough background checks.
No action taken, but increase monitoring of this trigger (or set of triggers).	Increase resources devoted to monitoring feedback from the field, in response to evidence from operations indicating a performance slowdown resulting from a rare combination of customer actions.
Further analysis needed; generate scenarios for a Survivable Systems Analysis (SSA) or carry out penetration testing to determine next activity, if any.	Create new scenarios that reflect the usage patterns of a new type of customer. Use these scenarios to perform an SSA, the results of which may drive additional evolutionary design activities.
Perform a portion (delta) of one or more survivability life-cycle activities.	A small change to the system architecture increases resistance to a new attack scenario.
Perform a portion (delta) of each of the full set of survivability life-cycle activities.	A modification to the mission touches all survivability life-cycle activities to some extent.
Abandon the system and do a full redesign.	A major change in the technology of the application domain, coupled with sweeping improvements in defensive technology, cannot be incorporated by evolutionary design activities alone.
Abandon the system.	A drastic change in the mission renders the system obsolete.

For example, a computer security expert on the SRT learns of a new attack technique that might threaten the survivability of the existing system. Assume that this new attack technique cannot be countered by straightforward maintenance activities such as applying a security patch to a system component or adding a new rule to the firewall. Based on the new attack technique, the security expert generates a set of new attack scenarios to be used as an input delta to a Survivable Systems Analysis (SSA) of the existing system. If deficiencies in the system's resistance, recognition, or recovery are discovered, then one or more survivability life-cycle activities (such as a modification of the system architecture, or a change in survivability requirements) will be required.

The completion of one survivability life-cycle activity may trigger the need for another. Adjustments in the design tradeoffs with other quality attributes in the system may also be called for. The point at which the evolutionary design process stops is dependent upon the risk tolerance of an organization, and the perceived cost–benefit ratio, with respect to the particular set of trigger events. If evolution is not feasible, the organization may tolerate the risk, or seek other alternatives that transcend the system.

It is imperative that the evolutionary design activities take place in the context of full access to a comprehensive set of artifacts of the design process (such as descriptions of the rationale for tradeoffs made during the last design cycle). Continuity of the design team is particularly crucial for the evolutionary design of survivable systems, so that the mission-specific design expertise can be sustained throughout the life of the system. Otherwise, the evolutionary design process may degenerate into a "patching" approach that can never support the long-term

survivability of systems. Just as survivability must be designed into a system from the beginning and not added on later as an afterthought, long-term survivability cannot be sustained through patching or routine maintenance, but only through the continual incorporation of new survivability solutions through a principled evolutionary design process.

5 COTS Development Life-Cycle Activities

Historically, computer security has been based on the use of a collection of generic tools and approaches that provide a fortress or perimeter defense for the applications being protected. For the most part, these security solutions were an add-on or afterthought. Moreover, the open, unbounded, highly vulnerable, highly collaborative Internet environment renders fortress models largely ineffective. Survivability may be thought of as a software engineering approach that integrates computer security into the software design and development process from the beginning. It protects the application-specific mission, provides recognition of problems that cannot totally be prevented, and provides recovery schemes when attacks (or accidents or component failures) cannot be completely avoided. Business risk management and engineering trade-offs are an inherent part of the development process for survivable systems.

Hence, survivability strategies must be integrated throughout the software development life cycle. This poses a particularly strong challenge for COTS-based software development. The implications for the development of survivable COTS-based systems are daunting. With the fortress or perimeter-based model of computer security, a COTS-based system could be developed with little or no regard to computer security concerns, and then a COTS-based perimeter defense (commercial firewalls plus a commercial encryption package, etc.) could be added to improve security. However, survivability is a global system property that emerges from the interactions among the system components and is difficult enough to discern when the internals of the components are completely known. With COTS, many of the software quality attributes are unknown and difficult to analyze without access to the source code or other artifacts of the software engineering process.

Our long-term goal is to create practical software development methodologies for building secure and survivable COTS-based systems. We plan to investigate two complementary areas of software engineering research: (1) survivability for traditional (i.e., custom) software development activities and (2) COTS-based system life-cycle activities.

Earlier in this report, we discussed our research work on development methodologies for survivable systems focused on the traditional software development life cycle and associated survivability activities. Unfortunately, these traditional life-cycle activities (augmented with survivability elements) cannot be directly applied to the development of COTS-based systems. As stated earlier, COTS-based systems pose special and severe challenges to any software development team. The foremost challenge is dealing with extremely limited information about the software quality attributes of the COTS products that are under consideration

for use as system components. Typically, none of the artifacts of the traditional software engineering process (source code, design rationale, test suites and test results, and so forth) are readily available. For a typical COTS product, it is therefore nearly impossible to discern the engineering tradeoffs that were made (explicitly or implicitly) among the various attributes of software quality (performance, security, reliability, maintainability, usability, etc.). Needless to say, this is a severe disadvantage for a development team that is trying to build a survivable system out of COTS components.

One useful step is an extension of the spiral life cycle model to incorporate survivability activities. This is shown in Figure 6.

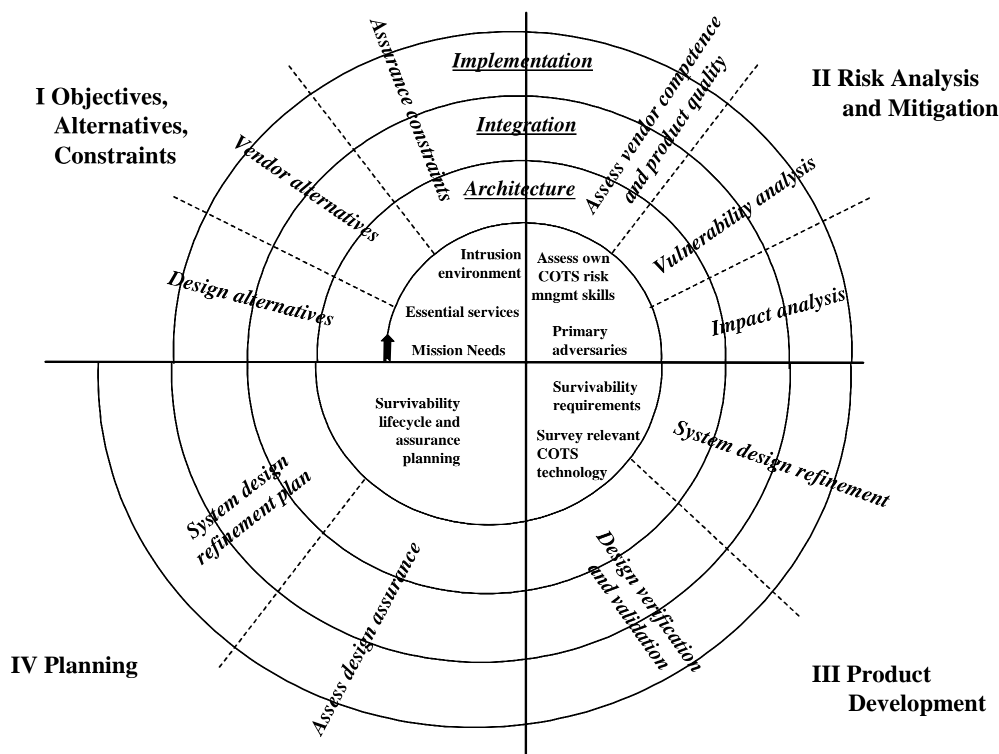

Figure 6: Spiral Life-Cycle Model with Survivability Activities

Imposing a principled software engineering process on the development of COTS-based systems has been the focus of earlier research on COTS-based system life-cycle activities [Brownsword 00, Oberndorf 00]. Although security and survivability concerns were not explicitly addressed in this earlier work, the research serves as a foundation (along with ongoing research on survivability for traditional life-cycle activities) for a development methodology for building survivable COTS-based systems.

The COTS-based system (CBS) life cycle includes four activity areas, each of which has several subareas [Oberndorf 00]:

- Engineering Activity Area: System Context, Architecture and Design, Marketplace, Construction, Configuration Management, Deployment and Support, Evaluation
- Business Activity Area: COTS Business Case, COTS Cost Estimation, Internal Supplier Relationships, Vendor Relationships
- Project-wide Activity Area: CBS Strategy, COTS Risk-Management, CBS Trade-offs, Information Sharing, Cultural Transition
- Contract Activity Area: Contract Requirements, Contract Tracking and Oversight, Solicitation, License Negotiation

Each subarea includes a set of activities. The complete set of activities is shown in Table 6. (Note that activities within an activity area are not sequential except where indicated.)

Table 6: CBS Life-Cycle Activities

Engineering Activity Area	**Activities**
System Context	Determine and prioritize negotiable and non-negotiable elements of the system context. Understand the essential elements of the end-users' business processes. Modify end-user processes. Negotiate system context changes. Reflect the results of trade-offs. Periodically reexamine business processes.
Architecture and Design	Select candidate products. Create and evolve architecture/design representation. Validate the architecture. Reflect results of trade-offs. Understand and reflect marketplace impact.
Marketplace	Create and maintain current knowledge of the available and emerging marketplace. Re-explore the marketplace. Alert technical staff to promising new technologies.
Construction (includes coding, integration, testing)	Discover and characterize product features. Create glue code. Integrate and test the system early and continuously. Continuously determine the effect of product upgrades.
Configuration Management	Identify configuration baselines. Receive and process upgrades. Systematically control changes. Release new system versions. Coordinate with construction.
Deployment and Support	Plan support. Plan system deployments.

	Plan and accommodate the need for end-user support. Incorporate new product releases. Coordinate with suppliers. Manage licenses. Perform site-specific tailoring. Plan and manage for multiple releases. Coordinate and engineer multiple supplier releases with your releases.
Evaluation	Plan evaluation. Design evaluation. Locate potentially relevant candidates. Perform appropriate analyses for selection of technologies or products. Document and share acquired information.
Business Activity Area	**Activities**
COTS Business Case (Note: These activities are sequential.)	Determine critical success factors for the system. Conduct a preliminary feasibility study. Identify key CBS assumptions. Articulate the alternatives. Formulate CBS strategic plans. Analyze CBS financial implications. Analyze alternatives. Revisit the COTS business case.
COTS Cost Estimation	Identify cost factors. Select and calibrate COTS cost-estimation models. Estimate costs. Provide cost estimates in support of other activity sets. Track actual costs vs. estimates. Maintain COTS cost-estimation models.
Internal Supplier Relationships	(written for government)
Vendor Relationships	Understand and monitor the vendor's long-term approach for maintenance and support. Develop a strategy to create and manage vendor relationships. Engage in meetings and exchanges with vendors. Establish liaisons for other vendor customers. Coordinate organization-wide vendor relationships. Encourage and facilitate working relationships among vendors.
Project-Wide Activity Area	**Activities**
CBS Strategy	Identify CBS goals, constraints, and assumptions. Identify COTS-related risks. Identify relevant market segments. Identify alternative COTS-based solutions. Assess, evaluate, and trade off alternative COTS-based solutions. Recommend an overall CBS strategy. Create a corresponding CBS plan. Reassess and revise the acquisition strategy and plan.

COTS Risk Management	Identify and prioritize COTS-related risks. Analyze COTS-related risks. Plan and institute COTS risk mitigations. Track COTS-related risks. Revisit CBS risk management success regularly.
CBS Trade-offs	Determine organization and contractor roles. Identify where CBS trade-offs are needed. Gather sufficient information to make informed COTS-related trade-offs. Select or make an appropriate CBS resolution. Communicate the resolution.
Information Sharing	Determine information collection and sharing strategies. Actively monitor the use of information you provided for sharing. Seek CBS information from outside sources. Ensure collection of CBS information. Make your CBS information readily accessible to others. Manage CBS information. Build information sharing into your processes.
Cultural Transition	Assess CBS readiness of your organization. Identify the skills sets needed for CBS success. Train everyone. Secure CBS buy-in of senior executives. Develop and implement a CBS cultural-transition strategy. Identify and encourage CBS champions. Provide incentives for changing. Share information.
Contract Activity Area	**Activities**
Contract Requirements (skills needed)	Address COTS-specific requirements in contract requirements. Appraise requests for contract changes to determine their effect on COTS products.
Contract Tracking and Oversight	Use testbeds and pilots to provide visibility. Involve the end-user community in pilots.
Solicitation	Prepare cost and schedule estimates for products. Prepare for the evaluation of responses. Conduct proposal evaluation.
License Negotiation	Conduct a preliminary investigation of licensing alternatives and costs. Secure a budget. Negotiate the licenses.

6 COTS Development Life-Cycle Activities and Survivability

For COTS-based systems (CBS), a survivability strategy can provide the framework for a specific set of survivability activities to be associated with the CBS life-cycle activities described previously in Table 6. The strategy should provide a framework for activities that are aimed at both process and product. For example, such a strategy should address policy as well as technical issues. Feasibility studies should be undertaken to determine whether COTS products can meet survivability requirements. Vendor evolution plans should be examined to determine whether the COTS products that currently meet survivability requirements will evolve in such a way as to continue to do so.

6.1 CBS Survivability Activities

We have made some modifications to the CBS activity areas and have populated a matrix with associated survivability activities [Mead 01]. The slightly revised CBS activity areas and subareas are

- Engineering Activity Area: System Context, Enterprise Architecture and Design, Marketplace, Construction, Configuration Management, Deployment and Sustainment, Evaluation
- Business Activity Area: COTS Business Case, COTS Cost Estimation, Vendor Relationships
- Project-Wide Activity Area: CBS Strategy, COTS Risk Management, CBS Trade-offs, Information Sharing, Cultural Transition, Policy
- Contract Activity Area: Contract Requirements (skills needed), Contract Tracking and Oversight, Solicitation, License Negotiation

A complete set of CBS activity areas supplemented with survivability activities is shown in Table 7.

Table 7: COTS Life-Cycle Activities Tailored to Survivability

Engineering Activity Area	Survivability Activities
System Context	Understand your overall business mission and its consequences in terms of survivability, survivability requirements, and essential services. Understand constraints such as existing networks, management issues, etc. Understand intrusion environment and potential intrusion scenarios. Understand survivability strategies of other systems external to this one. Periodically reexamine survivability context and requirements, and business processes as related to survivability, and trace the changes.
	Policy: Develop or modify overall policy to include survivability aspects.
Enterprise Architecture and Design	Refine overall survivability strategy in the architecture area. Use Survivable Systems Analysis, Survivable Network Design. Use survivability styles and strategies to guide architecture (Internal note: consider loose coupling and encapsulation). Understand survivability consequences of selected products. Incorporate survivability capabilities of selected products. Consider vulnerabilities outside specific components that may be part of the normal process, in both systems and operations.
	Policy: Consider the business processes that support technology and are essential for survivability.
Marketplace	Remain current on new survivability techniques. Revisit the marketplace with survivability in mind. Alert staff to survivability consequences and capabilities of new technologies.
Construction (includes coding, integration, testing)	Use defensive coding strategies, correctness verification, penetration testing, statistical testing. Continuously determine the impact of product upgrades on survivability. Consider integration and interoperability relative to survivability. Consider tailoring and its impact on survivability. Take preservation of properties into account. Develop a survivability argument.
	Policy
Configuration Management	Ensure that changes and upgrades do not negatively impact survivability. Use a configuration management scheme that will make survivability aspects visible.
	Policy
Deployment and Sustainment	Establish a survivability watch activity and a survivability risk assessment team. Consider vendor product evolution, technology evolution, and system evolution. Examine new products and new product releases for survivability. Look at long-term evolution and its survivability consequences; maintain and improve survivability. Adjust/react to technical decisions that partners and customers have made.
Evaluation	Assess the success of the survivability strategy. Perform survivability analyses for selection of technologies or products. Document and share acquired information.

Business Activity Area	Survivability Activities
COTS Business Case	Assess whether or to what extent COTS can support needed survivability features. Assess whether you can neutralize undesired side effects, e.g., automatic upgrades by the vendor. Assess duplication of effort/interoperability from a survivability viewpoint (e.g., does the vendor require separate password files that need separate maintenance?). Assess the cost impact of attacks. Determine critical survivability success factors for the system. Understand the financial implications and revisit the business case. Revisit the business case if critical sensitivity analysis factors change. This applies to both system and processes.
	Policy
COTS Cost Estimation	Use survivability as a cost factor in selected cost estimation models. Estimate cost impact of building in survivability or acquiring survivable COTS products for the threat environment.
Internal Supplier Relationships	
Vendor Relationships	Develop a strategy to assess vendors relative to survivability. Encourage and facilitate survivability discussions among vendors. How well-positioned are the vendors relative to where you want to go? Will they continue to be players in the long term?
Project-Wide Activity Area	**Survivability Activities**
CBS Strategy	Develop an overall survivability strategy. Develop a survivability plan. Identify needed survivability features. Examine alternative COTS-based solutions for survivability.
COTS Risk Management	Apply OCTAVE as part of the risk management scheme. Track top survivability risks in addition to overall system risks.
CBS Trade-offs	Assess survivability features of COTS products under consideration. Trade off survivability against required attributes in a project-wide context.
Information Sharing	Collect information about the threat environment and the survivability of CBS products and make it accessible to others. Include stakeholders and technical staff.
Cultural Transition	Identify survivability champions and ensure stakeholder buy-in to survivability needs. Provide awareness training on survivability to all personnel and in-depth training as needed. Keep in mind that the transition from security to survivability may be threatening to the traditional security staff, and obtain their participation and buy-in.

Contract Activity Area	Survivability Activities
Contract Requirements (skills needed)	Consider vendor and other contractor (e.g., integration, network) experience in survivability, and specify survivability in contract requirements.
Contract Tracking and Oversight	Get visibility into the survivability of COTS products. Have the ability to adjust contracts to reflect survivability changes, particularly relative to threats. Monitor COTS vendor performance relative to survivability requirements. Obligate contractors to share risks. Have the ability to audit/assess contractor systems.
Solicitation	Establish survivability evaluation criteria for vendor-provided products and services. Include survivability in cost and schedule estimates and in evaluation criteria.
License Negotiation	Determine whether survivability should be part of the license agreement. Evaluate survivability expectations in the event of expiration or change.

As an example, we provide some further detail on the System Context subarea.

6.2 System Context Survivability Issues

The ability to design and develop a survivable system depends in large measure on a thorough understanding of the context in which that system operates. The most salient characteristic of that context is the overall business mission that the system is designed to support. Ultimately it is the business mission that must survive, not any particular subsystem, component, or technology [Lipson 99]. Traditional computer security is based on a binary view of attack and defense, where an attack is either completely repelled or the attack succeeds and the system is compromised. In the open, highly-distributed, Internet environment of today, a perfect defense is impossible. In contrast, a survivable system degrades gracefully under attack, and continues to provide essential services even if one or more of its components is compromised. Survivability depends not only on a system's ability to resist attack, but also on its ability to recognize the effects of an attack, and its ability to recover from attacks that cannot be completely repelled. Elicitation of survivability requirements, which includes an enumeration and description of the essential services that a system must continue to provide in the face of attacks, is a critical early step in the development process.

System architects and engineers must also be aware of the contextual constraints imposed on the design by factors such as existing networks and technology that must function smoothly (or at least adequately) with the new system; management issues, including limitations on project funding, resources, and time to completion; and of course a finite set of existing COTS products from which to build the new survivable system. The use of widely available COTS products reduces costs, but their generic, mass-market, low-cost design makes it unlikely that such products will meet the specific survivability requirements of a particular application or system, particularly right out of the box. A crucial survivability activity, later in the design process, is to understand the survivability consequences of selected products. The

unavailability of source code, design rationale, and other engineering artifacts associated with COTS products makes them difficult to analyze for survivability, and for other attributes of software quality.

The threat environment and potential attack scenarios are additional aspects of the system context that must be considered in great detail. Survivability is a context-sensitive quality, and a system cannot be analyzed for survivability without an in-depth understanding of the mission, who and what would likely threaten that mission, and likely scenarios of the circumstances under which attacks on the system (and its mission) can be carried out. For example, a particular implementation of a cryptographic algorithm might be sufficient for protecting the daily transactions of a typical retailer on the Internet, but might be wholly inadequate to protect inter-bank transactions. Attack scenarios involving a banking system would posit a much greater use of resources by a potential attacker than would the scenarios for an attack on a low-profile retailer.

Another key aspect of the system context concerns the survivability strategies of external systems that the system depends on. Such external systems include local and global infrastructures. When a target system is heavily defended, a common attack strategy is to attempt to disrupt systems that provide services to the well-defended target. Hence, a survivability design strategy (and the risk-management analyses that support it) must consider the survivability of external systems, including those belonging to business partners, suppliers, customers, and collaborators. Survivability solution strategies should specify alternative means of obtaining the external services needed by the system, perhaps with a degraded but still acceptable quality of service.

Changes in a system's context are inevitable, and a survivable system must evolve over time to address those changes, or the survivability and security of the system will degrade. A periodic reexamination of the system context, including the underlying implementation technology, business mission, survivability requirements, and supporting policies and processes, is a critical part of the survivable system development life cycle. Addressing the results of this periodic reexamination is even more important for developing and sustaining the survivability of COTS-based systems. First, survivability strategies are based on the notion of an intelligent adversary, and so new vulnerabilities are continually being discovered in the underlying implementation technology, as are new means of exploiting such weaknesses. The pervasive distribution of many COTS-based systems makes them widely available for experimentation by hackers, and a COTS product vulnerability discovered in one system context can typically be exploited in many other systems that use that COTS product as a component. Second, the release schedule of a COTS component is typically not under the control of the design team of a COTS-based system. Upgrading quickly to the newest release may be a necessity to continue to meet some functional or non-functional system requirement, but the survivability implications of the upgrade will have to be evaluated (without the benefit of

engineering artifacts from the COTS component upgrade that would make the implications easier to discern).

Finally, a survivable system is dependent not only upon technology, but also upon the business policy and human processes that support the overall mission. System designers and developers must understand the policy context in which the system operates, or the survivability of the system will be at risk.

7 Future Research Opportunities

Many extensions of this work are possible. The larger context for survivability, system and COTS-based life-cycle models and their associated activities, could be investigated further. For example, one could expand upon and refine the CBS survivability activities, and reflect changes back to the CBS activity areas and subareas. Each of the activities could be described in much greater detail, along with examples and case studies, to provide a practical framework for building survivable CBS. This would allow CBS developers to begin to enhance and sustain system survivability. Our investigation of the survivability of CBS is part of an overall research activity into methods for survivable systems that can be incorporated into various life-cycle phases.

Some of the methods referenced in the activity areas are part of our research plan. A key next step for survivability evolution is to develop more powerful abstractions and reasoning methods for defining the behavior and structure of large-scale distributed systems. Such results will enable more comprehensive analysis of essential service and intrusion traces while limiting complexity. In addition, improved representations and methods are required for defining intrusions. It is important to move beyond the limitations of natural language and to develop uniform semantics for intrusion usage that permit more rigorous analysis and even allow for the application of computational methods.

Another fruitful line of research involves developing standardized architectural styles or templates for survivability strategies that can be inserted and composed with system architectures to improve their survivability properties. Such templates can be independently analyzed to define and document their contribution to system survivability.

References

[Anderson 97] Anderson, Robert H.; Hearn, Anthony C.; & Hundley, Richard O. “RAND Studies of Cyberspace Security Issues and the Concept of a U.S. Minimum Essential Information Infrastructure.” *Proceedings of the 1997 IEEE Information Survivability Workshop*. San Diego, California, Feb. 12-13, 1997. Los Alamitos, CA: IEEE Computer Society, 1997. <http://www.cert.org/research/isw.html>.

[Boehm 89] Boehm, Barry W. *Software Risk Management*. Washington, D.C.: IEEE Computer Society Press, 1989.

[Brownsword 00] Brownsword, Lisa; Oberndorf, Tricia; & Sledge, Carol A. “Developing New Processes for COTS-based Systems.” *IEEE Software 17*, 4 (July/August 2000): 48-55.

[Carrol 99] Carrol, John M. “Five Reasons for Scenario-Based Design.” *Proceedings of the Thirty-Second Annual Hawaii International Conference on Systems Sciences*. Maui, Hawaii, Jan. 5-8, 1999. Los Alamitos, CA: IEEE Computer Society Press, 1999.

[DoD 85] Department of Defense. “Department of Defense Trusted Computer System Evaluation Criteria.” DoD 5200.28-STD. National Computer Security Center, Department of Defense Computer Security Center, 1985.

[Ebert 97] Ebert, Christof. “Dealing with Nonfunctional Requirements in Large Software Systems.” *Annals of Software Engineering 3* (September 1997): 367-395.

[Ellison 98] Ellison, Robert; Fisher, David; Linger, Richard; Lipson, Howard; Longstaff, Thomas; & Mead, Nancy. “A Survivable Network Analysis Method.” *Proceedings of the 1998 IEEE Information Survivability Workshop*. Orlando, Florida, Oct. 28-30, 1998. Los Alamitos, CA: IEEE Computer Society, 1998. <http://www.cert.org/research/isw.html>.

[Ellison 99a] Ellison, Robert; Fisher, David; Linger, Richard; Lipson, Howard; Longstaff, Thomas; & Mead, Nancy. *Survivable Network Systems: An Emerging Discipline* (CMU/SEI-97-TR-013, ADA341963). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997 [revised 1999]. <http://www.sei.cmu.edu /publications/documents/97.reports/97tr013/97tr013abstract.html>.

[Ellison 99b] Ellison, Robert; Linger, Richard; Longstaff, Thomas; & Mead, Nancy. "Survivable Network System Analysis: A Case Study." *IEEE Software 16*, 4 (July/August 1999): 70-77.

[Kazman 98] Kazman, Rick; Klein, Mark; Barbacci, Mario; Longstaff, Thomas; Lipson, Howard; & Carriere, Jeromy. "The Architecture Tradeoff Analysis Method." *Proceedings of the International Conference on Engineering of Complex Computer Systems*. Monterey, CA, Aug. 10-14, 1998. Los Alamitos, CA: IEEE Computer Society Press, 1998. <http://www.sei.cmu.edu/ata/publications.html#papers>.

[Kemmerer 91] Kemmerer, R.A. & Porras, P.A. "Covert Flow Trees: A Visual Approach to Analyzing Covert Storage Channels." *IEEE Transactions on Software Engineering 17*, 11 (November 1991): 1166-1185.

[Linger 98] Linger, Richard; Mead, Nancy; & Lipson, Howard. "Requirements Definition for Survivable Network Systems." *Proceedings of the International Conference on Requirements Engineering*. Colorado Springs, CO, Apr. 6-10, 1998. Los Alamitos, CA: IEEE Computer Society Press, 1998. <http://www.cert.org/archive/pdf/icre.pdf>.

[Linger 99a] Linger, Richard. "Systematic Generation of Stochastic Diversity as an Intrusion Barrier in Survivable Systems Software." *Proceedings of the Thirty-Second Annual Hawaii International Conference on Systems Sciences*. Maui, Hawaii, Jan. 5-8, 1999. Los Alamitos, CA: IEEE Computer Society Press, 1999. <http://www.cert.org/archive/html/stochastic-divers.html>.

[Linger 99b] Linger, Richard & Trammell, Carmen J. "Cleanroom Software Engineering Theory and Practice," 351-372. *Industrial Strength Formal Methods in Practice*. Hinchey, Mike & Bowen, Jonathan, eds. London, UK: Springer-Verlag, 1999.

[Lipson 96] Lipson, Howard & Longstaff, Thomas. *Coming Attractions in Survivable Systems* (research report for DARPA). Pittsburgh, PA: CERT Coordination Center, Software Engineering Institute, Carnegie Mellon University, June 1996. <http://www.cert.org/research/>.

[Lipson 99] Lipson, Howard & Fisher, David A. "Survivability: A New Technical and Business Perspective on Security," 33-39. *Proceedings of the 1999 New Security Paradigms Workshop*. Caledon Hills, ON, September 22-24, 1999. New York, NY: Association for Computing Machinery, 2000.

[Marmor-Squires 88] Marmor-Squires, Ann & Rougeau, Pat. "Issues in Process Models and Integrated Environments for Trusted Systems Development," 109-113. *Proceedings of the 11th National Computer Security Conference*. Fort George G. Meade, MD, Oct. 17-20, 1988. Washington, D.C.: United States Government Printing Office, 1988.

[Marmor-Squires 89] Marmor-Squires, Ann; McHugh, John; Branstad, Martha; Danner, Bonnie; Nagy, Lou; Rougeau, Pat; & Sterne, Dan. "A Risk Driven Process Model for the Development of Trusted Systems," 184-192. *Proceedings of the 1989 Computer Security Applications Conference*. Tucson, AZ, Dec. 1989. Los Alamitos, CA: IEEE Computer Society Press, 1989.

[McHugh 84] McHugh, John. *Towards the Generation of Efficient Code From Verified Programs*. PhD Dissertation, The University of Texas at Austin, Austin, TX, 1984.

[McHugh 95] McHugh, John.; Payne, C.N.; & Martin, C. "Assurance Mappings," *Handbook for the Computer Security Certification of Trusted Systems*. NRL Technical Memorandum 5540:081A. Washington, DC: Naval Research Laboratory, January 1995.

[Mead 00a] Mead, Nancy; Ellison, Robert; Richard, Linger; Longstaff, Thomas; & McHugh, John. *Survivable Network Analysis Method* (CMU/SEI-2000-TR-013, ADA383771). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. <http://www.sei.cmu.edu/publications/documents/00.reports/00tr013.html>.

[Mead 00b] Mead, Nancy. "Lifecycle Considerations for Survivable Systems," *Cutter IT* E-Mail Advisor, July 26, 2000.

[Mead 00c] Mead, Nancy; Linger, Richard; McHugh, John; & Lipson, Howard. "Managing Software Development for Survivable Systems." *Annals of Software Engineering 11*, 1 (November 2001): 45-78.

[Mead 01] Mead, Nancy; Lipson, Howard; & Sledge, Carol. "Towards Survivable COTS-Based Systems." *Cutter IT Journal 14*, 2 (February 2001): 4-11.

[Mills 92] Mills, H.D. "Certifying the Correctness of Software." *Proceedings of 25th Hawaii International Conference on System Sciences*. Maui, Hawaii, Jan. 7-10, 1992. Los Alamitos, CA: IEEE Computer Society Press, 1992.

[Mills 97] Mills, Harlan D.; Linger, Richard C.; & Hevner, Alan R. *Principles of Information Systems Analysis and Design*. New York, NY: Academic Press, 1997.

[Oberndorf 00] Oberndorf, Tricia; Brownsword, Lisa; & Sledge, Carol A. *An Activity Framework for COTS-Based Systems* (CMU/SEI-2000-TR-010, ADA383836). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. <http://www.sei.cmu.edu /publications/documents/00.reports/00tr010.html>.

[Parnas 86] Parnas, D.L. & Clements, P.C. "A Rational Design Process: How and Why to Fake It." *IEEE Transactions on Software Engineering 12*, 2 (February 1986): 251-257.

[Prowell 99] Prowell, Stacy J.; Trammell, Carmen J.; Linger, Richard C.; & Poore, Jesse H. *Cleanroom Software Engineering: Technology and Process*. Boston, MA: Addison-Wesley, 1999.

[Royce 87] Royce, W.W, "Managing the Development of Large Software Systems." *Proceedings of the 9th International Conference on Software Engineering* Monterey, California, Mar. 30-Apr. 2, 1987. Los Alamitos, CA: IEEE Computer Society Press, 1987.

[Stavely 98] Stavely, Allan M. *Toward Zero-Defect Programming*. Boston, MA: Addison Wesley, 1998.

[Young 87] Young, W.D. & J. McHugh. “Coding for a Believable Specification to Implementation Mapping,” 140-149. *Proceedings of the 1987 IEEE Symposium on Security and Privacy*. Oakland, CA, Apr. 27-29, 1987. Los Alamitos, CA: IEEE Computer Society Press, 1987.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	October 2002	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Life-Cycle Models for Survivable Systems	F19628-00-C-0003

6. AUTHOR(S)

Richard C. Linger, Howard F. Lipson, John McHugh, Nancy R. Mead, Carol A. Sledge

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2002-TR-026

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	ESC-TR-2002-026

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Today's large-scale, highly distributed, networked systems improve the efficiency and effectiveness of organizations by permitting whole new levels of organizational integration. However, such integration is accompanied by elevated risks of intrusion and compromise. Incorporating survivability capabilities into an organization's systems can mitigate these risks. Current software development life-cycle models are not focused on creating survivable systems, and exhibit shortcomings when the goal is to develop systems with a high degree of assurance of survivability. If addressed at all, survivability issues are often relegated to a separate thread of project activity, with the result that survivability is treated as an add-on property. For each life-cycle activity, survivability goals should be addressed, and methods to ensure survivability incorporated.

This report explains survivability concepts, describes a software development life-cycle model for survivability, and illustrates techniques that can be applied during new development activities to support survivability goals. It also describes a software life-cycle model and associated activities to support survivability goals for systems based on commercial off-the-shelf products.

14. SUBJECT TERMS	15. NUMBER OF PAGES
software development life-cycle models, survivable systems, survivability	72

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL