Conceptual Model and Architecture of MAFTIA

A. Adelsbach, D. Alessandri, C. Cachin,
S. Creese, Y. Deswarte,K. Kursawe,
J. C. Laprie, D. Powell, B. Randell,
J. Riordan, P. Ryan, W. Simmonds, R. Stroud,
P. Verissimo, M. Waidner, A. Wespi

DI–FCUL TR–03–1

February 2003

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

Project IST-1999-11583

Malicious- and Accidental-Fault Tolerance for Internet Applications

Conceptual Model and Architecture of MAFTIA

David Powell[1] and Robert Stroud[2] (Editors)
[1]LAAS-CNRS, Toulouse
[2]University of Newcastle upon Tyne

MAFTIA deliverable D21

Public document

January 31, 2003

Technical Report CS-TR-787, University of Newcastle upon Tyne
Technical Report DI/FCUL TR-03-1, Universidade de Lisboa
LAAS-CNRS Report No. 03011
Research Report RZ 3473, IBM Research, Zurich Research Laboratory

List of Contributors[1]

André Adelsbach (Universität des Saarlandes)

Dominique Alessandri (IBM ZRL)

Christian Cachin (IBM ZRL)

Sadie Creese (Qinetiq)

Yves Deswarte (LAAS-CNRS)

Klause Kursawe (IBM ZRL)

Jean-Claude Laprie (LAAS-CNRS)

David Powell (LAAS-CNRS)

Brian Randell (University of Newcastle upon Tyne)

James Riordan (IBM ZRL)

Peter Ryan (University of Newcastle upon Tyne)

William Simmonds (Qinetiq)

Robert Stroud (University of Newcastle upon Tyne)

Paulo Veríssimo (Universidade de Lisboa)

Michael Waidner (IBM ZRL)

Andreas Wespi (IBM ZRL)

[1] Marc Dacier (Eurecom, France), Ian Welch (University of Wellington, New Zealand), and Birgit Pfitzmann (IBM, Zurich) are no longer members of the MAFTIA project, but contributed to an earlier version of this deliverable [Powell & Stroud 2001].

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

This deliverable builds on the work reported in [MAFTIA 2000] and [Powell & Stroud 2001]. It contains a further refinement of the MAFTIA conceptual model and a revised discussion of the MAFTIA architecture. It also introduces the work done in MAFTIA on verification and assessment of security properties, which is reported on in more detail in [Adelsbach & Creese 2003].

Chapter 2 is taken from [Avizienis et al. 2001] and presents core dependability concepts. This is a complete update with respect to [Powell & Stroud 2001]. It presents the latest version of the dependability concepts and gives a brief state of the art. This includes an analysis of the relationship between the terms dependability, survivability, and trustworthiness, all of which are seen to be essentially the same concept.

Chapter 3 refines the core dependability concepts in the context of malicious faults. The chapter now includes a completely new discussion of security policies and the relationship between security goals, properties, and rules. It is argued that a security failure only occurs if a security goal is violated, although violation of a security rule may lead the system into a state in which it is more liable to a security failure. There is also a discussion of the possible faults that can lead to security failures. The chapter continues by examining the distinction between intrusions, attacks, and vulnerabilities, and a taxonomy of different kinds of malicious logic has been added. There is also a discussion of how the traditional methods of building dependable systems, namely fault prevention, fault tolerance, fault removal, and fault forecasting, can be re-interpreted in a security context. The definition of an attack has been revised to distinguish between an attack as a human activity, and an attack as a technical activity, and as a result of this change, it is now possible to distinguish ten distinct security methods.

Chapter 4 introduces the topic of intrusion tolerance and shows how intrusion-detection systems relate to the traditional dependability notions of error detection and fault diagnosis. It goes on to present a framework for building intrusion-tolerant systems. The idea is that components in the overall system may be internally or externally monitored for erroneous behaviour. Some components may be intrusion-tolerant in that they can autonomously recover from detected errors. Detected errors are reported to a security administration component of the system that is responsible for diagnosis and managing intrusions at the system-wide level. A number of diagrams have been added to clarify intrusion detection concepts, and explain the role of the system security officer and the security subsystem in error detection, fault handling, and corrective maintenance.

Chapter 5 provides an overview of the MAFTIA architecture. It includes a discussion of the models and assumptions on which this architecture is based, together with an explanation of the various layers of the MAFTIA middleware and run-time support mechanism. There is also a description of the various intrusion-tolerance strategies that can be used to build intrusion-tolerant services. The chapter has been revised to explain the MAFTIA architecture in terms of the notion of trusted components that are only trusted to the extent of their trustworthiness. It is argued that this is an important new and innovative way of thinking about architectures for intrusion-tolerant systems, and the description of the MAFTIA architecture is now presented in these terms. The chapter is intended to summarise some of the key ideas underpinning the MAFTIA architecture, and thus serves as an introduction to some of the other deliverables, which go into more technical detail about these topics.

Chapter 6 discusses the formalisation of MAFTIA concepts and architectural principles, and introduces the work done on verification and assessment of secure systems, highlighting the novel contributions of MAFTIA in this area. In terms of the basic dependability concepts discussed in Chapter 3, the purpose of verification and assessment is vulnerability removal.

The chapter has been updated to reflect the latest results of this work, and also contains a substantial new section on issues surrounding the formalisation of security policies. The work on verification and assessment is discussed in much more detail in [Adelsbach & Creese 2003], which is based on previous work reported in [Adelsbach & Pfitzmann 2001, Adelsbach & Steiner 2002, Creese & Simmonds 2002].

Chapter 7 concludes the deliverable with a discussion of what has been achieved and a glossary of the terms used is given at the end of the report.

Chapter 2 Fundamental concepts of dependability[2]

Dependability is the system property that integrates such attributes as reliability, availability, safety, security, survivability, maintainability. The aim of this chapter is to summarize the fundamental concepts of dependability.

The protection and survival of complex information systems that are embedded in the infrastructure supporting advanced society has become a national and worldwide concern of the highest priority [Jones 2000]. Increasingly, individuals and organizations are developing or procuring sophisticated computing systems on whose services they need to place great reliance — whether to service a set of cash dispensers, control a satellite constellation, an airplane, a nuclear plant, or a radiation therapy device, or to maintain the confidentiality of a sensitive data base. In differing circumstances, the focus will be on differing properties of such services — e.g., on the average real-time response achieved, the likelihood of producing the required results, the ability to avoid failures that could be catastrophic to the system's environment, or the degree to which deliberate intrusions can be prevented. The notion of dependability provides a very convenient means of subsuming these various concerns within a single conceptual framework.

Our goal is to present a concise overview of the concepts, techniques and tools that have evolved over the past forty years in the field of dependable computing and fault tolerance. After a historical perspective, definitions of dependability are given. A structured view of dependability follows, according to a) the threats, i.e., faults, errors and failures, b) the attributes, and c) the means for dependability, namely fault prevention, fault tolerance, fault removal and fault forecasting.

2.1 Origins and integration of the concepts

The delivery of correct computing and communication services has been a concern of their providers and users since the earliest days. In the July 1834 issue of the *Edinburgh Review,* Dr. Dionysius Lardner published the article "Babbage's calculating engine", in which he wrote:

> *"The most certain and effectual check upon errors which arise in the process of computation, is to cause the same computations to be made by separate and independent computers; and this check is rendered still more decisive if they make their computations by different methods".*

The first generation of electronic computers (late 1940s to mid-50s) used rather unreliable components, therefore practical techniques were employed to improve their reliability, such as error control codes, duplexing with comparison, triplication with voting, diagnostics to locate failed components, etc. At the same time J. von Neumann, E. F. Moore and C. E. Shannon, and their successors developed theories of using redundancy to build reliable logic structures from less reliable components, whose faults were masked by the presence of multiple redundant components. The theories of masking redundancy were unified by W. H. Pierce as the concept of *failure tolerance* in 1965 (Academic Press). In 1967, A. Avizienis integrated masking with the practical techniques of error detection, fault diagnosis, and recovery into the concept of fault-tolerant systems [Avizienis 1967]. In the reliability modelling field, the major event was the introduction of the coverage concept by Bouricius, Carter and Schneider

---

[2] This chapter is available separately as the following report: A. Avizienis, J.-C. Laprie and B. Randell, *Fundamental Concepts of Dependability*, LAAS-CNRS, Research Report 01145, August 2001, Revision 1: December 2002 (UCLA CSD Report no. 010028; Newcastle University Report no. CS-TR-739). It is reproduced here with the authors' permission.

[Bouricius et al. 1969]. Seminal work on software fault tolerance was initiated by B. Randell [Randell 1975], and later complemented by work on N-version programming [Avizienis & Chen 1977].

The formation of the *IEEE-CS TC on Fault-Tolerant Computing* in 1970 and of *IFIP WG 10.4 Dependable Computing and Fault Tolerance* in 1980 accelerated the emergence of a consistent set of concepts and terminology. Seven position papers were presented in 1982 at FTCS-12 in a special session on fundamental concepts of fault tolerance, and J.-C. Laprie formulated a synthesis in 1985 [Laprie 1985]. Further work by members of IFIP WG 10.4, led by J.-C. Laprie, resulted in the 1992 book *Dependability: Basic Concepts and Terminology* (Springer-Verlag), in which the English text was also translated into French, German, Italian, and Japanese.

In this book, intentional faults (malicious logic, intrusions) were listed along with accidental faults (physical, design, or interaction faults). Exploratory research on the integration of fault tolerance and the defences against deliberately malicious faults, i.e., security threats, was started in the mid-80s [Dobson & Randell 1986, Fray et al. 1986]. The first IFIP Working Conference on Dependable Computing for Critical Applications was held in 1989. This and the six Working Conferences that followed fostered the interaction of the dependability and security communities, and advanced the integration of security (confidentiality, integrity and availability) into the framework of dependable computing.

2.2 The definitions of dependability

A systematic exposition of the concepts of dependability consists of three parts: the **threats** to, the **attributes** of, and the **means** by which dependability is attained, as shown in Figure 1.

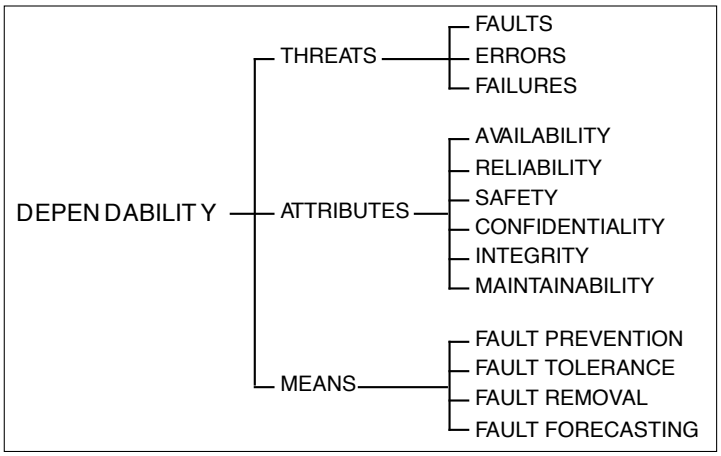


Figure 1 — The dependability tree

Computing systems are characterized by five fundamental properties: functionality, usability, performance, cost, and dependability. **Dependability** is the ability of a computing system to deliver service that can justifiably be trusted. The **service** delivered by a system is its behaviour, as perceived by its user(s); a **user** is another system (physical, human) that interacts with the former at the **service interface**. The **function** of a system is what the system is intended to do, and is described by the functional specification. **Correct service** is delivered when the service implements the system function. A system **failure** is an event that occurs when the delivered service deviates from correct service. A failure is thus a transition from correct service to **incorrect service**, i.e., to not implementing the system function. The delivery of incorrect service is a system **outage**. A transition from incorrect service to correct service is **service restoration**. Based on the definition of failure, an alternate definition of **dependability**, which complements the initial definition in providing a criterion for adjudicating whether the delivered service can be trusted or not, is as follows: the ability of a system to avoid failures that are more frequent or more severe, and outage durations that are

longer, than is acceptable to the user(s). In the opposite case, the system is no longer dependable: it suffers from a dependability failure.

2.3 The threats: faults, errors, and failures

A system may fail either because it does not comply with the specification, or because the specification did not adequately describe its function. An **error** is that part of the system state that may cause a subsequent failure: a failure occurs when an error reaches the service interface and alters the service. A **fault** is the adjudged or hypothesized cause of an error. A fault is **active** when it produces an error; otherwise it is **dormant**.

A system does not always fail in the same way. The ways in which a system can fail are its **failure modes**. These can be ranked according to **failure severities**. The modes characterize incorrect service according to four viewpoints:

- the failure domain,
- the controllability of failures,
- the consistency of failures, when a system has two or more users,
- the consequences of failures on the environment.

Figure 2 shows the modes of failures according to the above viewpoints, as well as failure symptoms that result from the combination of the domain, controllability and consistency viewpoints. The failure symptoms can be mapped into the failure severities as resulting from grading the consequences of failures.

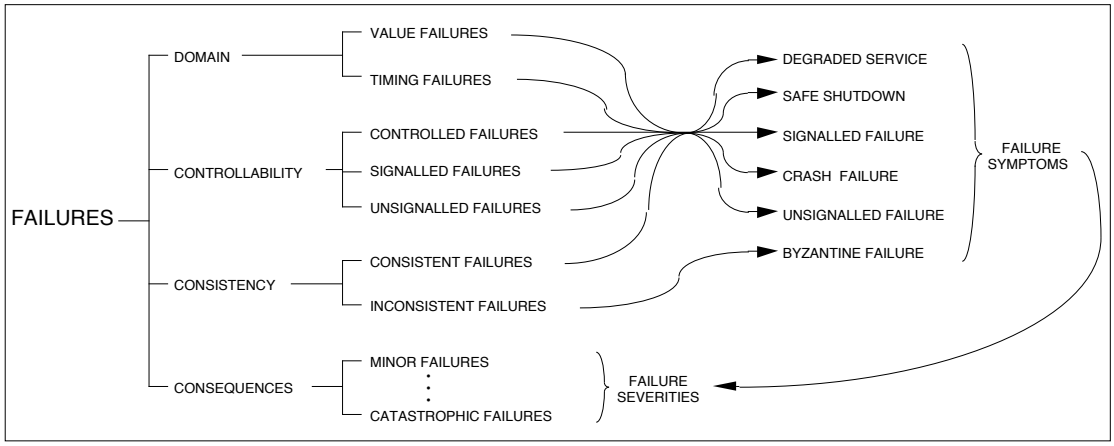


Figure 2 — The failure modes

A system consists of a set of interacting components, therefore the system state is the set of its component states. A fault originally causes an error within the state of one (or more) components, but system failure will not occur as long as the error does not reach the service interface of the system. A convenient classification of errors is to describe them in terms of the component failures that they cause, using the terminology of Figure 2: value vs. timing errors; consistent vs. inconsistent ('Byzantine') errors when the output goes to two or more components; errors of different severities: minor vs. ordinary vs. catastrophic errors. An error is **detected** if its presence is indicated by an **error message** or **error signal**. Errors that are present but not detected are **latent** errors.

Faults and their sources are very diverse. Their classification according to six major criteria is presented in Figure 3. It could be argued that introducing *phenomenological causes* in the classification criteria of faults may lead recursively to questions such as 'why do programmers make mistakes?', 'why do integrated circuits fail?' Fault is a concept that serves to stop recursion. Hence the definition given: *adjudged or hypothesized* cause of an error.

This cause may vary depending upon the viewpoint that is chosen: fault tolerance mechanisms, maintenance engineer, repair shop, developer, semiconductor physicist, etc.

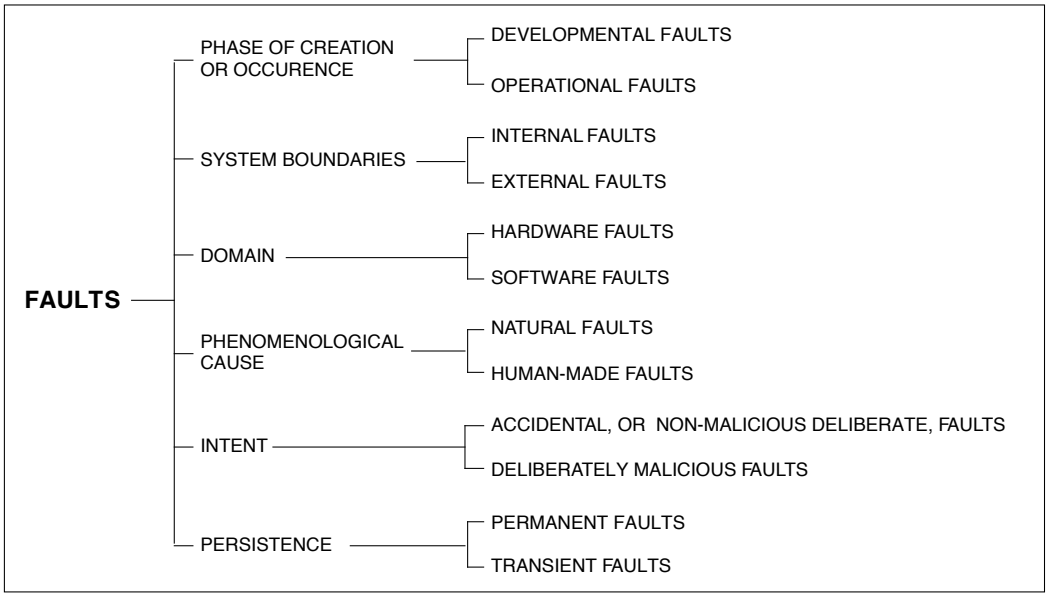


Figure 3 — Elementary fault classes

Combining the elementary fault classes of Figure 3 leads to the tree of the upper part of Figure 4. The leaves of the tree are gathered into three major fault classes for which defences need to be devised: **design faults**, **physical faults**, **interaction faults**. The boxes of Figure 4 point at generic illustrative examples of fault classes.

Non-malicious deliberate faults can arise during either development or operation. During development, they result generally from tradeoffs, either a) aimed at preserving acceptable performance and facilitating system utilization, or b) induced by economic considerations; such faults can be sources of security vulnerabilities, e.g., in the form of *covert channels*. Non-malicious deliberate interaction faults may result from the action of an operator either aimed at overcoming an unforeseen situation, or deliberately violating an operating procedure without having realized the possibly damaging consequences of his or her action. Non-malicious deliberate faults have the property that they are often only recognized as faults *after* an unacceptable system behaviour, thus a failure, has ensued; the specifier(s), designer(s), implementer(s) or operator(s) did not realize that the consequence of some decision of theirs was a fault.

Malicious faults affecting software fall into two classes: a) **malicious logics** [Landwehr et al. 1994], that encompass developmental faults such as *Trojan horses*, logic or timing *bombs*, and *trapdoors*, as well as operational faults (with respect to the given system) such as *viruses* or *worms*, and b) **intrusions**. There are interesting and obvious similarities between an attack that exploits a vulnerability (to cause an intrusion) and a physical external fault that ‘exploits’ a lack of shielding. It is in addition noteworthy that a) the external character of intrusions does not exclude the possibility that they may be attempted by system operators or administrators who are abusing their rights, and that b) attacks may use physical means to cause faults: power fluctuation, radiation, wire-tapping, etc.

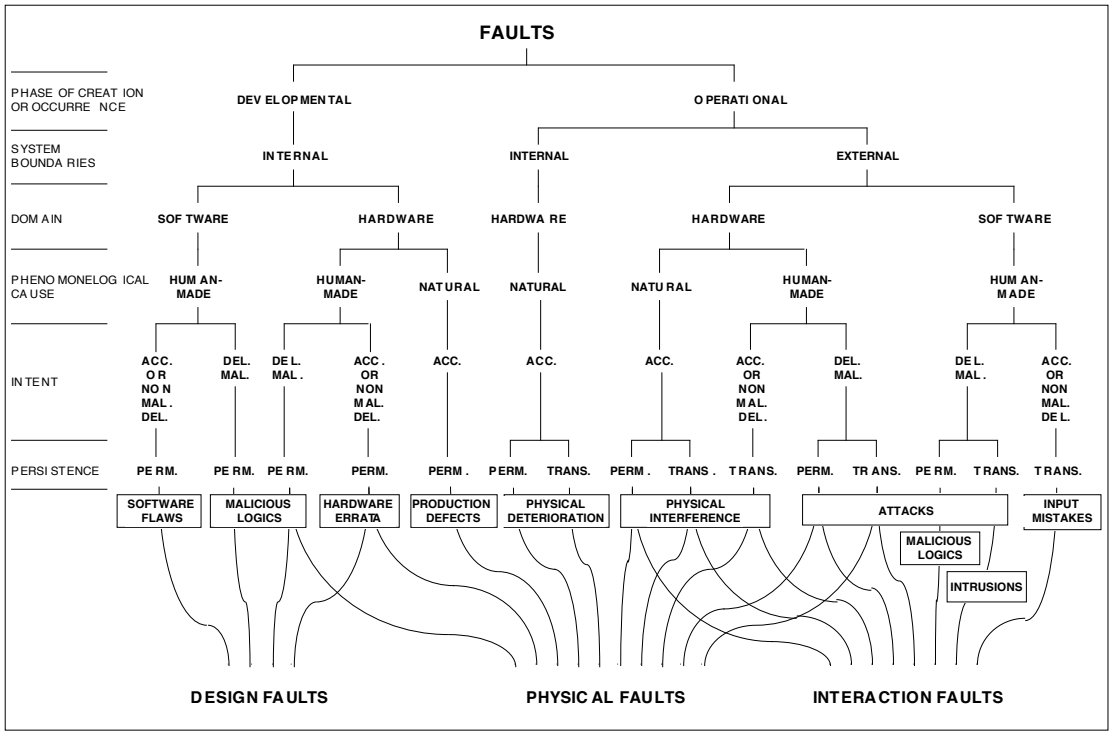


Figure 4 — Combined fault classes

Some design faults affecting software can cause so-called **software aging**, i.e., progressively accrued error conditions resulting in performance degradation or activation of elusive faults. Examples are memory bloating and leaking, unreleased file-locks, storage space fragmentation.

The relationship between faults, errors, and failures is addressed in Appendix A of this chapter, page 13.

Two final comments about the words, or *labels*, 'fault', 'error', and 'failure':

though we have chosen to use just these terms in this document, and to employ adjectives to distinguish different kinds of faults, errors and failures, we recognize the potential convenience of using words that designate, briefly and unambiguously, a specific class of threats; this is especially applicable to faults (e.g. bug, flaw, defect, deficiency, erratum) and to failures (e.g. breakdown, malfunction, denial-of-service);

the semantics of the terms fault, error, failure reflect current usage: i) fault prevention, tolerance, and diagnosis, ii) error detection and correction, iii) failure rate, failure mode.

2.4 The attributes of dependability

Dependability is an integrative concept that encompasses the following basic attributes:

- **availability**: readiness for correct service,
- **reliability**: continuity of correct service,
- **safety**: absence of catastrophic consequences on the user(s) and the environment,
- **confidentiality**: absence of unauthorized disclosure of information,
- **integrity**: absence of improper system state alterations,
- **maintainability**: ability to undergo repairs and modifications.

Depending on the application(s) intended for the system, different emphasis may be put on different attributes. The description of the required goals of the dependability attributes in terms of the acceptable frequency and severity of the failure modes, and of the corresponding acceptable outage durations (when relevant), for a stated set of faults, in a stated environment, is the **dependability requirement** of the system.

Several other dependability attributes have been defined that are either combinations or specializations of the six basic attributes listed above. **Security** is the concurrent existence of a) availability for authorized users only, b) confidentiality, and c) integrity with 'improper' taken as meaning 'unauthorized'. Characterizing a system's reaction to faults is of special interest, via, e.g., **robustness**, i.e., dependability with respect to erroneous inputs.

The attributes of dependability may be emphasized to a greater or lesser extent depending on the application: availability is always required, although to a varying degree, whereas reliability, safety, confidentiality may or may not be required. The extent to which a system possesses the attributes of dependability should be interpreted in a relative, probabilistic, sense, and not in an absolute, deterministic sense: due to the unavoidable presence or occurrence of faults, systems are never totally available, reliable, safe, or secure.

Integrity is a prerequisite for availability, reliability and safety, but may not be so for confidentiality (for instance, attacks via covert channels or passive listening can lead to a loss of confidentiality, without impairing integrity). The definition given for integrity – absence of improper system state alterations – extends the usual definition as follows: (a) when a system implements an authorization policy, 'improper' encompasses 'unauthorized'; (b) 'improper alterations' encompass actions that prevent (correct) upgrades of information; (c) 'system state' encompasses hardware modifications or damages. The definition given for maintainability goes beyond corrective and preventive maintenance, and encompasses forms of maintenance aimed at adapting or perfecting the system.

Security has not been introduced as a single attribute of dependability. This is in agreement with the usual definitions of security, which view it as a composite notion, namely 'the combination of (1) confidentiality (the prevention of the unauthorized disclosure of information), (2) integrity (the prevention of the unauthorized amendment or deletion of information), and (3) availability (the prevention of the unauthorized withholding of information)' [ITSEC]. A single definition for **security** could be: the absence of unauthorized access to, or handling of, system state.

Besides the attributes defined at the beginning of the section, and discussed above, other, *secondary*, attributes can be defined. An example of such a secondary attribute is **robustness**, i.e., dependability with respect to external faults, which characterizes a system's reaction to a specific class of faults. The notion of secondary attributes is especially relevant for security, when we distinguish among various types of information. Examples of such secondary attributes are:

- **accountability**: availability and integrity of the identity of the person who performed an operation,
- **authenticity**: integrity of a message content and origin, and possibly of some other information, such as the time of emission,
- **non-repudiability**: availability and integrity of the identity of the sender of a message (non-repudiation of the origin), or of the receiver (non-repudiation of reception).

Variations in the emphasis on the different attributes of dependability directly affect the appropriate balance of the techniques (fault prevention, tolerance, removal and forecasting) to be employed in order to make the resulting system dependable. This problem is all the more difficult as some of the attributes conflict (e.g. availability and safety, availability and security), necessitating design trade-offs.

Other concepts similar to dependability exist, such as survivability and trustworthiness. They are presented in Appendix B of this chapter, page 15.

2.5 The means to attain dependability

The development of a dependable computing system calls for the combined utilization of a set of four techniques:

- **fault prevention**: how to prevent the occurrence or introduction of faults,
- **fault tolerance**: how to deliver correct service in the presence of faults,
- **fault removal**: how to reduce the number or severity of faults,
- **fault forecasting**: how to estimate the present number, the future incidence, and the likely consequences of faults.

The concepts relating to these techniques are presented in this section. A brief state-of-the-art is presented in Appendix C of this chapter, page 16

2.5.1 Fault prevention

Fault prevention is attained by quality control techniques employed during the design and manufacturing of hardware and software. Such techniques include structured programming, information hiding, modularization, etc., for software, and rigorous design rules for hardware. Shielding, radiation hardening, etc., intend to prevent operational physical faults, while training, rigorous procedures for maintenance, 'foolproof' packages, intend to prevent interaction faults. Firewalls and similar defences intend to prevent malicious faults.

2.5.2 Fault tolerance

Fault tolerance is intended to preserve the delivery of correct service in the presence of active faults. It is generally implemented by error detection and subsequent system recovery.

Error detection originates an error signal or message within the system. An error that is present but not detected is a **latent** error. There exist two classes of error detection techniques:

- **concurrent error detection**, which takes place during service delivery,
- **preemptive error detection**, which takes place while service delivery is suspended; it checks the system for latent errors and dormant faults.

Recovery transforms a system state that contains one or more errors and (possibly) faults into a state without detected errors and faults that can be activated again. Recovery consists of error handling and fault handling. **Error handling** eliminates errors from the system state. It may take three forms:

- **rollback**, where the state transformation consists of returning the system back to a saved state that existed prior to error detection; that saved state is a **checkpoint**,
- **compensation**, where the erroneous state contains enough redundancy to enable error elimination,
- **rollforward**, where the state without detected errors is a new state.

Fault handling prevents faults from being activated again. Fault handling involves four steps:

- **fault diagnosis**, which identifies and records the cause(s) of error(s), in terms of both location and type,
- **fault isolation**, which performs physical or logical exclusion of the faulty components from further participation in service delivery, i.e., it makes the fault dormant,

- **system reconfiguration**, which either switches in spare components or reassigns tasks among non-failed components,
- **system reinitialization**, which checks, updates and records the new configuration and updates system tables and records.

Usually, fault handling is followed by corrective maintenance that removes faults isolated by fault handling. The factor that distinguishes fault tolerance from maintenance is that maintenance requires the participation of an external agent.

Systematic usage of compensation may allow recovery without explicit error detection. This form of recovery is called **fault masking**. However, such simple masking will conceal a possibly progressive and eventually fatal loss of protective redundancy; thus, practical implementations of masking generally involve error detection (and possibly fault handling).

Preemptive error detection and handling (often called BIST: built-in self-test), possibly followed by fault handling is often performed at system power up. It may also come into play during operation, under various forms such as spare checking, memory scrubbing, audit programs, or so-called software rejuvenation, aimed at removing the effects of software aging before they lead to failure.

Fail-controlled systems are designed and implemented so that they fail only in specific modes of failure described in the dependability requirement and only to an acceptable extent, e.g., with stuck output as opposed to delivering erratic values, silence as opposed to babbling, consistent as opposed to inconsistent failures. A system whose failures are, to an acceptable extent, halting failures only is a **fail-halt**, or **fail-silent**, **system**. A system whose failures are, to an acceptable extent, all minor ones is a **fail-safe system**.

The choice of error detection, error handling and fault handling techniques, and of their implementation, is directly related to the underlying fault assumption. The classes of faults that can actually be tolerated depend on the fault assumption in the development process. A widely-used method of fault tolerance is to perform multiple computations in multiple channels, either sequentially or concurrently. When tolerance of operational physical faults is required, the channels may be of identical design, based on the assumption that hardware components fail independently. Such an approach has proven to be adequate for elusive design faults, e.g., via rollback, however it is not suitable for the tolerance of solid design faults, which necessitates that the channels implement the same function via separate designs and implementations, i.e., through **design diversity** [Avizienis & Chen 1977, Randell 1975].

Fault tolerance is a recursive concept: it is essential that the mechanisms that implement fault tolerance should be protected against the faults that might affect them. Examples are voter replication, self-checking checkers, ‘stable’ memory for recovery programs and data, etc. Systematic introduction of fault tolerance is facilitated by the addition of support systems specialized for fault tolerance such as software monitors, service processors, dedicated communication links.

Fault tolerance is not restricted to accidental faults. Some mechanisms of error detection are directed towards both malicious and accidental faults (e.g., memory access protection techniques) and schemes have been proposed for the tolerance of both intrusions and physical faults, via information fragmentation and dispersal [Fray et al. 1986], as well as for tolerance of malicious logic, and more specifically of viruses, either via control flow checking, or via design diversity [Joseph & Avizienis 1988].

2.5.3 Fault removal

Fault removal is performed both during the development phase, and during the operational life of a system.

Fault removal during the development phase of a system life-cycle consists of three steps: verification, diagnosis, correction. **Verification** is the process of checking whether the system

adheres to given properties, termed the verification conditions. If it does not, the other two steps follow: diagnosing the fault(s) that prevented the verification conditions from being fulfilled, and then performing the necessary corrections. After correction, the verification process should be repeated in order to check that fault removal had no undesired consequences; the verification performed at this stage is usually termed non-regression verification.

Checking the specification is usually referred to as **validation**. Uncovering specification faults can happen at any stage of the development, either during the specification phase itself, or during subsequent phases when evidence is found that the system will not implement its function, or that the implementation cannot be achieved in a cost effective way.

Verification techniques can be classified according to whether or not they involve exercising the system. Verifying a system without actual execution is **static verification**, via static analysis (e.g., inspections or walk-through), model-checking, theorem proving. Verifying a system through exercising it constitutes **dynamic verification**; the inputs supplied to the system can be either symbolic inputs in the case of **symbolic execution**, or actual inputs in the case of verification testing, usually simply termed **testing**. An important aspect is the verification of fault tolerance mechanisms, especially a) formal static verification, and b) testing that necessitates faults or errors to be part of the test patterns, a technique that is usually referred to as **fault injection**. Verifying that the system cannot do more than what is specified is especially important with respect to what the system should not do, thus with respect to safety and security. Designing a system in order to facilitate its verification is termed **design for verifiability**. This approach is well-developed for hardware with respect to physical faults, where the corresponding techniques are termed design for testability.

Fault removal during the operational life of a system is corrective or preventive maintenance. **Corrective maintenance** is aimed at removing faults that have produced one or more errors and have been reported, while **preventive maintenance** is aimed to uncover and remove faults before they might cause errors during normal operation. The latter faults include a) physical faults that have occurred since the last preventive maintenance actions, and b) design faults that have led to errors in other similar systems. Corrective maintenance for design faults is usually performed in stages: the fault may be first isolated (e.g., by a workaround or a patch) before the actual removal is completed. These forms of maintenance apply to non-fault-tolerant systems as well as fault-tolerant systems, as the latter can be maintainable on-line (without interrupting service delivery) or off-line (during service outage).

2.5.4 Fault forecasting

Fault forecasting is conducted by performing an evaluation of the system behaviour with respect to fault occurrence or activation. Evaluation has two aspects:

- **qualitative**, or **ordinal**, **evaluation**, which aims to identify, classify, rank the failure modes, or the event combinations (component failures or environmental conditions) that would lead to system failures,
- **quantitative**, or **probabilistic**, **evaluation**, which aims to evaluate in terms of probabilities the extent to which some of the attributes of dependability are satisfied; those attributes are then viewed as measures of dependability.

The methods for qualitative and quantitative evaluation are either specific (e.g., failure mode and effect analysis for qualitative evaluation, or Markov chains and stochastic Petri nets for quantitative evaluation), or they can be used to perform both forms of evaluation (e.g., reliability block diagrams, fault-trees).

The evolution of dependability over a system's life-cycle is characterized by the notions of stability, growth, and decrease that can be stated for the various attributes of dependability. These notions are illustrated by **failure intensity**, i.e., the number of failures per unit of time.

It is a measure of the frequency of system failures, as noticed by its user(s). Failure intensity typically first decreases (reliability growth), then stabilizes (stable reliability) after a certain period of operation, then increases (reliability decrease), and the cycle resumes.

The alternation of correct-incorrect service delivery is quantified to define reliability, availability and maintainability as measures of dependability:

- **reliability**: a measure of the continuous delivery of correct service — or, equivalently, of the time to failure,
- **availability:** a measure of the delivery of correct service with respect to the alternation of correct and incorrect service,
- **maintainability**: a measure of the time to service restoration since the last failure occurrence, or equivalently, measure of the continuous delivery of incorrect service,
- **safety** is an extension of reliability: when the state of correct service and the states of incorrect service due to non-catastrophic failure are grouped into a safe state (in the sense of being free from catastrophic damage, not from danger), **safety** is a measure of continuous safeness, or equivalently, of the time to catastrophic failure; safety is thus reliability with respect to catastrophic failures.

Generally, a system delivers several services, and there are often two or more modes of service quality, e.g. ranging from full capacity to emergency service. These modes distinguish less and less complete service deliveries. Performance-related measures of dependability are usually subsumed into the notion of **performability**.

The two main approaches to probabilistic fault-forecasting, aimed at deriving probabilistic estimates of dependability measures, are modelling and (evaluation) testing. These approaches are complementary, since modelling needs data on the basic processes modelled (failure process, maintenance process, system activation process, etc.), which may be obtained either by testing, or by the processing of failure data.

When evaluating fault-tolerant systems, the effectiveness of error and fault handling mechanisms, i.e. their coverage, has a drastic influence on dependability measures [Bouricius et al. 1969]. The evaluation of coverage can be performed either through modelling or through testing, i.e., **fault injection**.

2.6 Conclusion

A major strength of the dependability concept, as it is formulated in this chapter, is its integrative nature, that enables a unification of the more classical notions of reliability, availability, safety, security, maintainability, that are then seen as attributes of dependability. The fault-error-failure model is central to the understanding and mastering of the various threats that may affect a system, and it enables a unified presentation of these threats, while preserving their specificities via the various fault classes that can be defined. Equally important is the use of a fully general notion of failure as opposed to one that is restricted in some way to particular types, causes or consequences of failure. The model provided for the means for dependability is extremely useful, as those means are much more orthogonal to each other than the more classical classification according to the attributes of dependability, with respect to which the design of any real system has to perform trade-offs due to the fact that these attributes tend to conflict with each other.

Appendix A The pathology of failure: relationship between faults, errors and failures

The creation and manifestation mechanisms of faults, errors, and failures are illustrated by Figure 5, and summarized as follows:

1. A fault is active when it produces an error, otherwise it is dormant. An active fault is either a) an internal fault that was previously dormant and that has been activated by the computation process or environmental conditions, or b) an external fault. Fault activation is the application of an input (the activation pattern) to a component that causes a dormant fault to become active. Most internal faults cycle between their dormant and active states.
2. Error propagation within a given component (i.e., *internal* propagation) is caused by the computation process: an error is successively transformed into other errors. Error propagation from one component (C1) to another component (C2) that receives service from C1 (i.e., *external* propagation) occurs when, through internal propagation, an error reaches the service interface of component C1. At this time, service delivered by C1 to C2 becomes incorrect, and the ensuing failure of C1 appears as an external fault to C2 and propagates the error into C2. The presence of an error within a system can arise from a) the activation of an internal fault, previously dormant, b) the occurrence of a physical operational fault, either internal or external, or c) the propagation of an error from another system (interacting with the given system) via the service interface, i.e., an **input error**.
3. A failure occurs when an error is propagated to the service interface and unacceptably alters the service delivered by the system. A failure of a component causes a permanent or transient fault in the system that contains the component. Failure of a system causes a permanent or transient external fault for the other system(s) that interact with the given system.

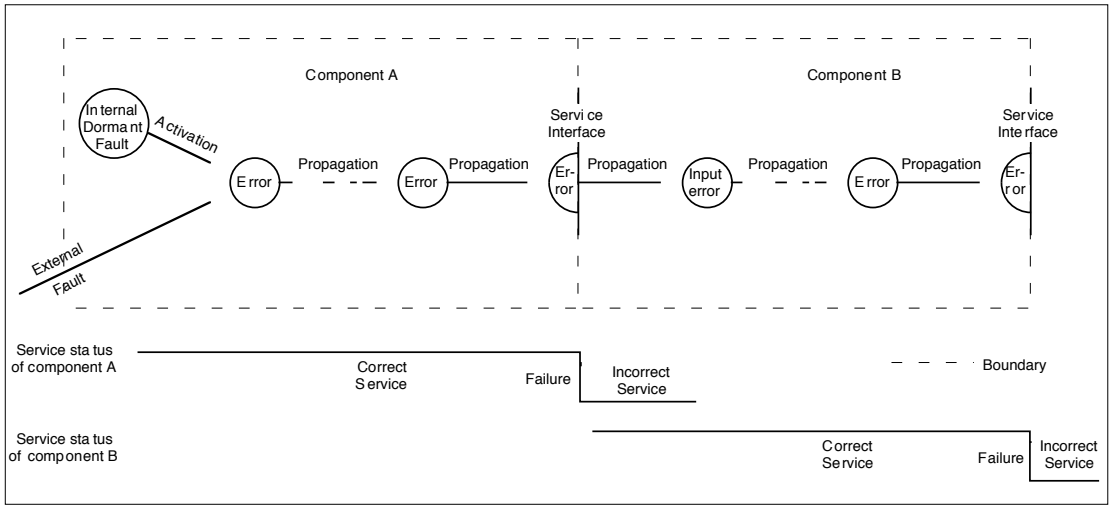


Figure 5 — Error propagation

These mechanisms enable the 'fundamental chain' to be completed, as indicated by Figure 6.

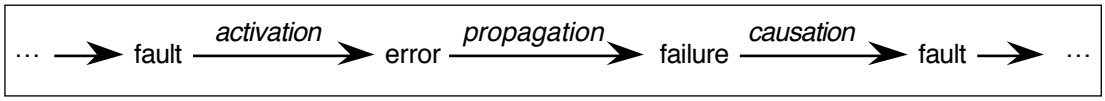


Figure 6 — The fundamental chain of dependability threats

The arrows in this chain express a causality relationship between faults, errors and failures. They should be interpreted generically: by propagation, several errors can be generated before a failure occurs.

Some illustrative examples of fault pathology are given in Table 1. From those examples, it is easily understood that fault dormancy may vary considerably, depending upon the fault, the given system's utilization, etc.

Table 1 — Examples illustrating fault pathology

- The result of an error by a programmer leads to a failure to write the correct instruction or data, which in turn results in a *(dormant) fault* in the written software (faulty instruction(s) or data); upon activation (invoking the component where the fault resides and triggering the faulty instruction, instruction sequence or data by an appropriate input pattern) the fault becomes *active* and produces an error; if and when the error affects the delivered service (in information content and/or in the timing of delivery), a *failure* occurs. This example is not restricted to accidental faults: a logic bomb is created by a malicious programmer; it will remain dormant until activated (e.g. at some predetermined date); it then produces an error that may lead to a storage overflow or to slowing down the program execution; as a consequence, service delivery will suffer from a so-called denial-of-service.
- The result of an error by a specifier leads to a failure to describe a function, which in turn results in a *fault* in the written specification, e.g. incomplete description of the function. The implemented system therefore does not incorporate the missing (sub-)function. When the input data are such that the service corresponding to the missing function should be delivered, the actual service delivered will be different from expected service, i.e., an *error* will be perceived by the user, and a *failure* will thus occur.
- An inappropriate human-system interaction performed by an operator during the operation of the system is an external *fault* (from the system viewpoint); the resulting altered processed data is an *error;* etc.
- An error in reasoning leads to a maintenance or operating manual writer's failure to write correct directives, which in turn results in a *fault* in the corresponding manual (faulty directives) that will remain dormant as long as the directives are not acted upon in order to address a given situation, etc.

The ability to identify the activation pattern of a fault that caused one or more errors is the **fault activation reproducibility**. Faults can be categorized according to their activation reproducibility: faults whose activation is reproducible are called **solid**, or **hard**, faults, whereas faults whose activation is not systematically reproducible are **elusive**, or **soft**, faults. Most residual design faults in large and complex software are elusive faults: they are intricate enough that their activation conditions depend on complex combinations of internal state and external requests that occur rarely and can be very difficult to reproduce. Other examples of elusive faults are:

- 'pattern sensitive' faults in semiconductor memories, changes in the parameters of a hardware component (effects of temperature variation, delay in timing due to parasitic capacitance, etc.);
- error conditions — affecting either hardware or software — that occur when the system load exceeds a certain level, causing, e.g., marginal timing and synchronization.

The similarity of the manifestation of elusive design faults and of transient physical faults leads to both classes being grouped together as **intermittent faults**. Errors produced by intermittent faults are usually termed **soft errors** [Bossen & Hsiao 1980].

The complexity of the mechanisms of creation, activation and manifestation of faults leads to the possibility of several causes. Such complexity leads to the definition, whether for classifying faults uncovered during operation or for stating fault hypotheses during system

design, of classes of faults that are more general than the specific classes considered up to now, in the sense that they may in turn have physical, design or interaction causes, or combinations of those. An example of such a class of faults is the **configuration change faults**: service delivery is altered during adaptive or perfective maintenance operations performed on-line, concurrently with system operation (e.g., introduction of a new software version on a network server).

Appendix B Dependability, survivability, trustworthiness: three names for an essential property

The protection of highly complex societal infrastructures controlled by embedded information systems against all classes of faults defined in the section on the threats to dependability, including intelligent attacks, has become a top priority of governments, businesses, and system builders. As a consequence, different names have been given to the same essential property that assures protection. Here we compare the definitions of three widely known concepts: dependability, survivability, and trustworthiness (Table 2).

Table 2 — Dependability, survivability and trustworthiness

Concept	Dependability	Survivability	Trustworthiness
Goal	1) ability to deliver service that can justifiably be trusted 2) ability of a system to avoid failures that are more frequent or more severe, and outage durations that are longer, than is acceptable to the user(s)	capability of a system to fulfil its mission in a timely manner	assurance that a system will perform as expected
Threats present	1) design faults (e.g., software flaws, hardware errata, malicious logics) 2) physical faults (e.g., production defects, physical deterioration) 3) interaction faults (e.g., physical interference, input mistakes, attacks, including viruses, worms, intrusions)	1) attacks (e.g., intrusions, probes, denials of service) 2) failures (internally generated events due to, e.g., software design errors, hardware degradation, human errors, corrupted data) 3) accidents (externally generated events such as natural disasters)	1) hostile attacks (from hackers or insiders) 2) environmental disruptions (accidental disruptions, either man-made or natural) 3) human and operator errors (e.g., software flaws, mistakes by human operators)
Reference	This chapter	“Survivable network systems” [Ellison et al. 1997]	“Trust in cyberspace” [Schneider 1999]

A side-by-side comparison leads to the conclusion that all three concepts are essentially equivalent in their goals and address similar threats. Trustworthiness omits the explicit listing of internal faults, although its goal implies that they also must be considered. Such faults are implicitly considered in survivability via the (component) failures. Survivability was present in the late sixties in the military standards, where it was defined as a system capacity to resist hostile environments so that the system can fulfill its mission (see, e.g., MIL-STD-721 or DOD-D-5000.3); it was redefined recently, as described below. Trustworthiness was used in a study sponsored by the National Research Council, referenced below. One difference must be noted. Survivability and trustworthiness have the threats explicitly listed in the definitions, while both definitions of dependability leave the choice open: the threats can be either all the

faults of Figure 4 (page 4), or a selected subset of them, e.g., ‘dependability with respect to design faults’, etc.

Appendix C Where do we stand?

Advances in integration, performance, and interconnection, are the elements of a virtuous spiral that led to the current state-of-the-art in computer and communication systems. However, two factors at least would tend to turn this virtuous spiral into a deadly one without dependability techniques: a) a decreasing natural robustness (due to, e.g., an increased sensitivity of hardware to environmental perturbations), and b) the inevitability of residual faults that goes along with the increased complexity of both hardware and software.

The interest in **fault tolerance** grows accordingly to our dependence in computer and communication systems. There exists a vast body of results, from error detecting and correcting codes to distributed algorithms for consensus in the presence of faults. Fault tolerance has been implemented in many systems; Table 3 lists some existing systems, either high availability systems or safety-critical systems.

Table 3 — Examples of fault tolerant systems

		Tolerance to design faults
High availability platforms	Stratus [Wilson 1985]	—
	Tandem SeverNet [Baker et al. 1995]	elusive software design faults
	IBM S/390 cluster [Brown Associates 1998]	—
	Sun cluster [Bowen et al. 1997]	—
Safety-critical systems	SACEM Subway speed control [Hennebert & Guiho 1993]	hardware design faults and compiler faults
	ELEKTRA Railway signalling system [Kantz & Koza 1995]	hardware and software design faults
	Airbus Flight Control System [Brière & Traverse 1993]	hardware and software design faults
	Boeing 777 Flight Control System [Yeh 1998]	hardware design faults and compiler faults

The progressive mastering of physical faults has enabled a dramatic improvement of computing systems dependability: the overall mean time to failure or unavailability has decreased by two orders of magnitude over the last two decades. As a consequence, design and human-made interaction faults currently dominate as sources of failure, as exemplified by Table 4. Tolerance of those two classes of faults is still an active research domain, even if approaches have been devised and implemented, especially for software design faults.

Table 4 — Proportions of failures due to accidental or non-malicious deliberate faults
(consequences and outage durations are highly application-dependent)

	Computer systems (e.g., transaction processing [Gray 1990], electronic switching [Cramp et al. 1992])		Larger, controlled systems (e.g., commercial airplanes [Ruegger 1990], telephone network [Kuhn 1997])	
	Rank	Proportion of failures	Rank	Proportion of failures
Physical internal faults	3	~ 10 %	2	15 - 20 %
Physical external faults	3	~ 10 %	2	15 - 20 %
Human-made interaction faults	2	~ 20 %	1	40 - -50 %
Design faults	1	~ 60 %	2	15 - 20 %

Security has benefited from advances in cryptography (e.g., public key schemes) and in the policies aimed at controlling information flow. Although there is much less in the way of published statistics about malicious faults than there is about accidental faults, a recent survey by Ernst & Young estimated that on the average two-thirds of the 1200 companies surveyed in 32 countries suffered from at least one fraud per year, and that 84% of the frauds were perpetrated by employees. System security policies are the current bottleneck, be it due to the vulnerabilities induced by the inevitable residual design faults in their implementation mechanisms, or by the necessary laxity without which systems would not be operable. Fault tolerance is thus needed, for protection from both malicious logics and intrusions.

The cost of verification and validation of a computing system is at least half of its development cost, and can go as high as three quarters for highly critical systems. The dominance in these costs of **fault removal** explains the importance of research in verification: the density of faults created during software development lies in the range of 10 to 300 faults/kLoC (thousand lines of executable code), down to 0.01 to 10 faults/kLoC after fault removal. The latter are residual faults, persisting during operation, and such high densities explain the importance of failures induced by design faults in large software systems.

Significant advances have taken place in both static and dynamic verification. Table 5 lists various current model checking and theorem proving tools. The current bottleneck lies in the applicability to large-scale systems, and thus to scalability from critical embedded systems to service infrastructures.

Table 5 — Examples of theorem proving and model-checking tools

Model checking	Theorem proving
Design/CPN (http://www.daimi.aau.dk/designCPN/)	Coq Proof Assistant (http://coq.inria.fr)
FDR (http://www.formal.demon.co.uk/FDR2.html)	HOL (http://lal.cs.byu.edu/lal/hol-desc.html)
HyTech (http://www.eecs.berkeley.edu/~tah/HyTech)	Isabelle (http://www.cl.cam.ac.uk/Research/HVG/Isabelle/)
KRONOS (http://www-verimag.imag.fr/TEMPORISE/kronos/)	PVS (http://www.csl.sri.com/pvs.html)
NuSMV (http://sra.itc.it/tools/nusmv/index.html)	
SPIN (http://netlib.bell-labs.com/netlib/spin)	
UPPAAL (http://www.docs.uu.se/docs/rtmv/uppaal/)	

The difficulty, or even the impossibility, of removing all faults from a system leads naturally to **fault forecasting**. Powerful probabilistic approaches have been developed, widely used as far as physical operational faults are concerned. Evaluating dependability with respect to software design faults is still a controversial topic, especially for highly-critical systems. Table 6 lists various current software tools for stable dependability evaluation. Tools for reliability growth evaluation are described in the *Handbook of software reliability engineering* edited by M. Lyu (McGraw-Hill, 1996). Only recently has the use of probabilistic evaluation of security started to gain acceptance.

Table 6 — Examples of software tools for dependability evaluation

Stable dependability	
Tool name	Type of models
SHARPE (http://www.ee.duke.edu/~chirel/research1)	Fault trees and Markov chains
SPNP (http://www.ee.duke.edu/~chirel/research1)	Stochastic Petri nets and Markov chains
SURF-2 (http://www.laas.fr/laasef/index.htm)	Stochastic Petri nets and Markov chains
GreatSPN (http://www.di.unito.it/~greatspn/)	Stochastic Petri nets and Markov chains
Ultra-SAN (http://www.crhc.uiuc.edu/PERFORM/UltraSAN)	Stochastic Activity Networks
DSPNexpress (http://www4.cs.uni-dortmund.de/~Lindemann/software/DSPNexpress/mainE.html)	Deterministic and stochastic Petri nets

Complementarity of fault removal and fault forecasting is yet more evident for fault tolerant systems for which verification and validation must, in addition to functional properties, address the ability of those systems to deal with faults and errors. Such verifications involve fault injection. Table 7 lists various current fault-injection tools.

Table 7 — Examples of fault-injection tools

Hardware, pin level, fault injection	Software implemented fault injection	Simulators with fault injection capability
RIFLE (http://eden.dei.uc.pt:80/~henrique/Informatica/Rifle.html)	FERRARI (http://www.cerc.utexas.edu/~jaa/ftc/fault-injection.html) Xception (http://eden.dei.uc.pt:80/~henrique/Informatica/Xception.html) MAFALDA (http://www.laas.fr/laasve/index.htm)	DEPEND (http://www.crhc.uiuc.edu/DEPEND/depend.html) MEFISTO (http://www.laas.fr/laasve/index.htm) VERIFY (http://www3.informatik.uni-erlangen.de/Projects/verify.html)

Confining within acceptable limits development costs of systems of ever-growing complexity necessitates development approaches involving pre-existing components, either acquired (COTS, commercial off-the-shelf software, or OSS, open source software) or re-used. The dependability of such components may be inadequate or simply unknown, which leads to the necessity of characterizing their dependability before using them in a development, or even to improve their dependability via wrapping them.

Improving the state of the art in dependable computing is crucial for the very future of Sciences and Technologies of Information and Communication. Computer failures cost society tens of billions of dollars each year, and the cumulative cost of cancelled software developments is of the same order of magnitude (http://standishgroup.com/visitor/chaos.htm). Significant actions have been undertaken, such as the *High Confidence Software and Systems* component of the Federal Research Programme on Technologies of Information, or the *Cross-Programme Action on Dependability* of the European Information Society Technology Programme.

Chapter 3 Refinement of core concepts with respect to malicious faults

In this chapter we elaborate on the core dependability concepts with respect to malicious faults and generalise towards security in general. These generalisations constitute our contribution towards a unified terminology for the security domain and are intended to complement existing work such as the glossary initiated by the NSA (National Security Agency) [NSA 1998].

3.1 Security policies

The security literature is somewhat confused as to the exact nature of security policies and their role in the design and maintenance of a secure system. A security policy may vary from a "set of managerial platitudes", to borrow Anderson's phrase [Anderson 2001], at one extreme to something quite rigorous like a Role Based Access Control (RBAC) policy at the other [Ferraiolo & Kuhn 1992]. Furthermore, a security policy typically spans several levels of abstraction: at the highest level it states the security requirements in the form of goals, at lower levels it might stipulate particular mechanisms and implementations and detailed rules such as minimal length of keys and passwords. A security policy will also typically span both the technical and social aspects of the system.

We will think of a security policy as comprising both goals and rules and we draw a distinction between these. Goals are intended to capture the high-level security requirements and as such any violation of the goals constitutes a security failure. The rules will typically be lower level constraints on the system behaviour that are designed to ensure that the system is robust against (possibly malicious) faults. Violations of the rules will typically not correspond to security failures but correspond to erroneous states in which the system is more prone to failure.

Ideally, a security policy should be so structured that the rules imply the goals. That is, conformity with the rules guarantees that the goals are upheld. In practice, things tend not to be so simple: some security goals may be difficult or impossible to reduce to technical rules and socio-technical mechanisms are required to induce users to maintain the spirit of the policy. Sometimes it may be simpler to express a goal directly in terms of rules. Thus the distinction between goals and rules may not be entirely clear-cut.

3.1.1 Security goals

We will interpret the security goals or requirements of a system as being high-level statements of what security properties the system must guarantee. The violation of a security goal corresponds to a security failure of the system. Typical security goals might include: "the confidentiality of sensitive data must be maintained", or "the integrity and availability of system data to authorised users must be maintained".

As they stand such statements have very little meaning until we have made precise the meanings of the various terms. We will discuss shortly the definition of the basic security properties such as: *confidentiality, availability, integrity* and so on. The security policy will also serve to define what data is regarded as *sensitive* and which users are regarded as *authorised* to exercise which privileges.

3.1.2 Security rules

Rules often take the form of constraints on the system states and transitions. By "system" here we mean in the broad socio-technical sense; embracing the human users as well as the purely

technical aspects (computers, networks etc). Some of the rules will apply to the users and take the form of prohibitions, duties and obligations. The rules that serve to constrain the behaviour of the technical system will usually take the form of access or privilege control rules and we will refer to such rules as "technical rules". Rules that constrain the human users will be referred to as "social rules". Where we mean the ensemble of rules, both technical and social, we will use the term "rules".

In a well-structured security policy, the rules should imply the goals, i.e., adherence to the rules should result in the system maintaining the goals. However the rules will typically be stronger than the goals. That is, the rules will be sufficient but not necessary. Violation of a rule will not necessarily, of itself, constitute a security failure but may render the system more liable to a security failure. Thus, for example, a rule may stipulate that passwords must be random and of minimal length 8 characters. Violation of such a rule does not immediately constitute a security failure but does make the system more liable to a guessing or dictionary attack on the password. In the terminology of dependability, we see that the violation of such a rule is best thought of as corresponding to an error.

The arguments that allow us to conclude that a set of rules imply certain security goals typically involve various architectural and modelling assumptions and approximations. For example, we may wish to argue that certain access controls imply that sensitive information will not be available to unauthorised users. The argument will depend on numerous assumptions about the strength of mechanism (e.g., that 8 random characters is sufficient for passwords, 128 bit AES is effectively unbreakable etc.), possible access modes and non-bypassability of the access control mechanisms.

3.1.3 Security properties

Here we give informal descriptions of the principal security properties.

3.1.3.1 Confidentiality

Confidentiality (sometimes referred to as secrecy) is the property that unauthorised users do not acquire knowledge of sensitive information. It is usually expressed in terms of appropriate controls on information flows. The policy will specify what flows are allowed and which are forbidden, i.e., which users may have sight of which data items.

There is no equivalent in traditional dependability of "loss of confidentiality". Moreover, loss of confidentiality may be very difficult to detect. Loss of confidentiality means that some information is present in a place (computer storage, paper, some user's brain…) where it should not be. This is an error, i.e., "part of the system state which may lead to a failure", the failure being the disclosure of this information outside the considered system. This error, at least theoretically, can be detected: the state of the (socio-technical) system is different to what it would be in the absence of error, so comparison of different copies should detect that. Moreover, if "undetachable" labels are attached to items of information, it may be possible to detect that this information should not be where it is, e.g., presence of information labelled as secret in an unclassified file (the labels form a sort of error-detecting code). As for other errors, a confidentiality error can be inferred by a detected failure (e.g., publication of the confidential information in a newspaper). The failure may remain undetected (the information is propagated to unauthorised users, but not published) in the same way as other failures can remain undetected for a long time (e.g., the Therac system).

Another issue with loss of confidentiality is the inherent difficulty of recovery. Since a lost secret cannot become secret again, the only apparent way to recover confidentiality is to substitute a new secret for the lost secret, e.g., by asking a user to choose a new password (i.e., a form of forward recovery). This can be done pre-emptively in order to limit the duration of the threat posed by a compromised secret. For some forms of information it may be extremely difficult to recover from breaches of confidentiality. Once sensitive medical history information has leaked it cannot simply be updated, for example.

We describe, in Chapter 6 of this document, how security models allow us to formally define information flows in terms of an appropriate underlying model of computation or interaction, in particular how we can characterise the absence of information flow over a particular channel or interface.

3.1.3.2 Integrity

In the most general and abstract terms, we can think of data integrity as being the property that the data remain "valid" in an appropriate sense. Sometimes this notion of validity may be in terms of a notion of internal consistency, in which case it can be specified purely in terms of the state of the target system. An example of this might be the requirement to keep books balanced in a financial application. Sometimes it means rather more: that the data accurately reflect an external reality. In a medical application, for example, this might mean that a patient record accurately represents the patient's medical history. Such a definition cannot be given purely in terms of the consistency of the system state and requires reference to the state of the environment.

In practice, integrity properties are often replaced by operational constraints: records can only be created and modified by authorised users. This shifts the problem of defining the "validity" of data to one of depending on (i.e., trusting) the honesty and competence of the authorised users. This interpretation of integrity can be viewed as being a dual of confidentiality. Thus, confidentiality requires that sensitive information not flow to unauthorised users, whilst integrity requires that unauthorised users not influence (integrity) sensitive information. Thus, the direction in which information flow is forbidden is flipped.

Sometimes these operational constraints are elaborated to include stipulations that access be only through certain "well-formed" functions (or methods in an object-oriented context). A well-formed function here is defined to be one that is guaranteed to preserve the appropriate internal consistency requirement. Thus, for banking applications, well-formed accounting functions will ensure that the books remain balanced. The Clark-Wilson model provides a framework for describing such mechanisms [Clark & Wilson 1987]. Such mechanisms can help maintain internal consistency requirements but are little help with external consistency requirements.

3.1.3.3 Availability

Confidentiality and integrity are, roughly speaking, about preventing undesirable events occurring. Availability on the other hand is about ensuring that desirable events do occur. When a user requests a service for which they are authorised, the system should ensure that this service is provided in both a correct and timely fashion. There are thus two aspects to an availability property: the value domain and the time domain.

The timeliness aspect of availability is sometimes defined as: the system will provide the service *eventually*. The problem with such a definition is that a system can never be deemed in finite time to have failed against such a requirement (though if the system returns an incorrect value it will have failed against the correctness aspect of the requirement). As a result, availability is sometimes defined in terms of some finite response time or quality of service requirement.

3.1.3.4 Other security properties

Many security properties can be defined in terms of the confidentiality, integrity and availability of the information or the service itself, or of some meta-information[3] related to the information or service. Examples of such meta-information include:

---

[3] Of course, at some level (e.g., at the operating system level), meta-information might be embodied as "real" information.

- time of a service delivery, or of creation, modification or destruction of an item of information;
- identity of the person who invoked an operation: creator of an item of information, author of a document, sender or receiver of an item of information, etc.;
- location or address of an item of information, a communication entity, a device, etc.;
- existence of an item of information or a service;
- existence of an information transfer, or a communication channel, or of a message, etc.;
- occurrence of an operation;
- sensitivity level of an item of information or meta-information;
- certainty or plausibility level of an item of information or meta-information;

For example, an *accountability* property [CEN 13608-1, ISO 7498-2, Trouessin 2000] can be expressed in terms of the availability and integrity of a set of meta-information about the existence of an operation, the identity of the person who realised the operation, the time of the operation, etc. *Anonymity* is the confidentiality of the identity of the person, for instance, who invoked an operation. *Privacy* is confidentiality with respect to personal data, which can be either "information" (such as the content of a registration database), or "meta-information" such as the identity of a user who performed a particular operation, sent a particular message, received the message, etc.

Traffic analysis is an attack against the confidentiality of communication meta-information, to gain knowledge of the existence of a channel, of the existence of a message, of the identities, locations or addresses of the message sender and receiver, of the time of a communication, etc.

Authenticity is the property of being "genuine". For a message, authenticity is equivalent to integrity of both the message content (information integrity) and of the message origin, and possibly of other meta-information such as time of emission, classification level, etc. (meta-information integrity). In the same manner, a document is authentic if its content has not been altered (information integrity) and optionally if the declared author is the real author and not a plagiarist, if the publication date is correct, etc. (meta-information integrity). In the same way, an alleged user is authentic if the declared identity is the real identity of that person. *Authentication* is the process that gives confidence in authenticity.

Non-repudiation corresponds to the availability and authenticity of some meta-information, such as creator identity (and possibly time of creation) for non-repudiation of origin, or reception and receiver identity for non-repudiation of reception.

Many security properties can be expressed in terms of the availability, integrity and confidentiality properties applied to information and meta-information.

3.1.3.5 Discussion

Some remarks are in order regarding the nature of certain security properties. These become important when we come to map them onto the MAFTIA conceptual model. Many security properties turn out not to be trace properties (also known as safety properties). That is to say, they cannot be stated as predicates over behaviours (traces) of the target system. A trace of a system S can be thought of as a record of certain observables of a particular execution of S. We will use the terms "a trace" and "an execution" interchangeably. We do not need to be precise at this stage about the form that this record takes: it could be a record of a sequence of actions or states or indeed a combination.

A trace/safety property defines those behaviours that are regarded as acceptable or safe, and so can be expressed as a set of traces. To verify that a given system S satisfies a given safety property, it suffices to determine that the trace set $\tau(S)$ of S is a subset of the set of safe traces.

To check that a system maintains a safety property at run time, we can simply monitor its actions and check that the trace remains in the set of safe traces. As soon as an action *a* occurs that takes the trace outside the set of allowed traces, the property will have been violated and the action *a* will be regarded as a failure.

It is useful at this point to introduce the concept of an Execution Monitor (EM) [Schneider 2000]. An EM is a conceptual device that runs in parallel with the system, observing the (relevant aspects of the) system behaviour and blocking any action of the target system that would result in a violation of the trace property in question. Given an arbitrary system *S* and arbitrary trace property P we can define an EM that will constrain *S*'s behaviours and ensure that the composed system satisfies *P*.

Note that an EM has no-look ahead or look-sideways capability. That is to say, the determination of what actions are allowed after a given execution depends only on the trace up to that point and cannot depend on information about possible future behaviour or possible alternative behaviours. In particular, the EM has no model of the system or environment that might inform its decisions as to what actions to block. Such devices might be interesting to investigate, but they fall outside the class of execution monitors that we are considering here.

Trace properties therefore have a special significance:

- They can be defined in terms of individual executions
- They can be enforced by an execution monitor
- Violations have a clear cut characterisation at a well defined point in an execution.
- They are preserved by standard refinement relations. In particular, if a trace *t* satisfies the property, then so will any prefix of *t*.

By contrast, some of the basic security properties fail to be trace properties.

For example, the timeliness aspect of availability cannot be captured as a trace property, and a violation cannot even be detected by an EM, let alone enforced. If we observe an execution of S, there will never be a point in the execution at which we can say that the availability property has been violated. We cannot determine that a system will never perform a certain action by observing it for some finite time: there will always be the possibility that the required action will be performed at some later time.

On the other hand, if we adopt a time-limited notion of availability, we can express this as a trace property and detect a violation at run time: If during some execution a service is requested but not provided within the requisite time limit then, as soon as the time limit is reached, the availability property will be deemed to have been violated.

Note that even though time-limited availability can be characterised as a trace property, it still cannot be enforced by an execution monitor. An execution monitor, by definition, can only block actions and cannot force the target system to perform an action.

Information flow properties, such as confidentiality and some flavours of integrity, represent yet a further class of property: they are not trace properties, nor are they availability properties. We can characterise an availability failure with respect to a given execution if we assume a god-like ability to the possible future unfoldings of the execution. In this case we can determine that a certain action will never be available. More realistically, if we assume a detailed understanding of the construction of the system we may be able to determine that, having reached a certain state, it will never be able to execute a certain action.

For information flow properties, even the ability to foresee the future is insufficient to characterise a failure given only the state of the target system. Information flow properties are

characteristics of the set of possible system behaviours, not of individual behaviours. To establish whether a system will maintain confidentiality we must look at the set of all its possible behaviours and establish whether this set as a whole has certain characteristics. An information flow property can be formulated as a predicate over the space of processes or alternatively as a predicate over the power set of the set of traces (as opposed to a predicate over the set of traces).

To make this clearer, consider a simple example: consider a system that takes in sensitive plaintext and outputs this encrypted with a one-time-pad over an open channel. This will be perfectly secure as long as the system really does implement a one-time-pad, i.e., is able to generate any of the possible key streams (unpredictably and with equal probability). If the system malfunctions in such a way that it will in fact only generate one of a small number of the possible key streams, this would constitute a security failure but would not be detectable by observing a single execution.

Again, it is useful to turn to the concept of an execution monitor. We cannot, in general, find an execution monitor that can constrain a given system in such a way as to ensure that it will satisfy a given information flow property. An information flow property has to be designed in at the outset. To return to the stream cipher example, the algorithm has to be designed and verified to be effectively indistinguishable from a random process. No execution monitor will convert a flawed crypto device into a secure one.

For an information flow property, we typically cannot observe an individual run of the target system and identify a point at which this behaviour has violated the property. Of course, we can broaden our notion of the system to include the states of the adversary. If we do this then a breach of confidentiality can be characterised by the appearance in the adversary's state of some sensitive data item. This is fine from a conceptual standpoint, and indeed is the approach often taken in modelling and analysis (see Chapter 6, for example). In reality, however, we cannot expect to have (direct) access to the adversary's state of knowledge.

The history of cryptography is full of examples of security failures that went undetected. Of course, if the security failure never manifests itself in the form of tangible damage to the system then we could argue that it is irrelevant. The problem is that the damage may manifest itself with an unbounded latency and possibly in some entirely different system. An additional difficulty is that diagnosing the cause of the damage to the security breach may be difficult. Thus, during World War II, the Germans seem never to have realised that the Enigma cipher has been broken by the Allies. The Allies were always careful to ensure that alternative explanations of U-boat sinking would be possible, for example by sending over a "fortuitous" spotter plane.

The upshot of this discussion is that care must be taken in interpreting non-trace properties in the standard conceptual model of dependable systems. The notion of a failure as usually used in dependability fits most naturally with trace-based properties. Failures with respect to non-trace properties can be characterised in a conceptual sense but may not be manifest from observation of the target system. As a result, failures with respect to information flow properties may, in principle, be hard to detect and so may propagate across system boundaries and layers.

We see that properties can fall into a number of categories. They may be:

- enforceable in the Schneider sense, e.g., a trace property,
- not enforceable in the Schneider sense but a violation may be immediately and locally detectable, e.g., a time-bounded availability property,
- not enforceable and violations may not be immediately detectable, e.g., an information flow property.

A property for which violations cannot be detected will not be enforceable.

We see that there are strong theoretical as well as pragmatic reasons, in true dependability style, to deploy a mixture of prevention (enforcement), detection, tolerance, recovery and reconfiguration.

3.2 Security failures and their causes

In this section we discuss a number of security failures and their possible causes, i.e. faults.

3.2.1 Faults in the specification of the security goals

First we note that problems may arise in capturing the security requirement of the system in question. We have to articulate the requirement as the security policy in some informal or, better still, formal specification. Requirements capture is a notoriously slippery process and mistakes may easily arise at this stage. Furthermore, even if our initial formulation of the security requirements is perfectly accurate, requirements creep and evolution of the threat environment, etc., may render our initial formulation inappropriate over time.

It may also be that certain goals are in conflict: i.e., cannot be (fully) satisfied simultaneously. For example, the requirement to maintain patient privacy may conflict with the requirement to make medical information available for research purposes. The requirement for anonymity of sperm and egg donors runs counter to the right-to-know of the offspring. It may also be that security goals conflict with non-security goals. Thus, for example, the requirement to keep a patient's records confidential may conflict (in certain circumstances) with the requirement to do everything possible to ensure the life and health of the patient.

We therefore have a class of security failures due to inaccurate capture of the security requirements. This of course applies equally to non-security requirements.

3.2.2 Faults in the specification of the security rules

Let us suppose that we have succeeded in formulating the security goals in such a way that they do accurately capture the security requirements. We discuss now the ways in which failures with respect to these goals may occur.

Firstly we note that it may be that the rules as specified, both technical and socio-technical, fail to imply the goals. This may arise in a number of ways:

- The rules, social and technical, may be incomplete.
- The rules may be inconsistent or ambiguous.
- Mistakes may be made in the analysis of the logical consequences of a given rule set.

In general, security goals must be enforced by a mixture of technical and social mechanisms. Whether or not the security goals are upheld will depend on the behaviour of the users as well as that of the technical system. To some extent, user behaviour can be constrained by technical means: namely those user actions that require the cooperation of the technical system, accessing a sensitive file, for example.

However, many user actions do not require system cooperation. For example, a user may disclose sensitive information orally to an unauthorised user, and so cannot be technically constrained. These should be covered by duties and obligations: users are under an obligation not to divulge sensitive information.

There is a further class of actions for which, though they do involve human-machine interaction, it may be difficult to determine whether the action is in accordance with the goals. For example, it is difficult by purely technical means to detect a violation of the obligation on a clinician to maintain accurate patient records. The system will typically not be able to determine that an update to the record is due or determine whether an update is correct.

This is an example of a class of security failure referred to as an abuse of privilege: where a goal is violated without any violation of the technical mechanisms. Abuses of privilege are discussed further in Section 3.3.5.

The *principle of least privilege* seeks to minimise the scope for abuse of privilege by ensuring that only the minimal privileges needed to fulfil duties and tasks are allocated. There are theoretical and practical limits on the extent to which the security goals can be technically enforced. We have noted in Section 3.1.3.5 some of the origins of the theoretical limits to the extent to which security properties may be technically enforced. There are also practical limitations on the extent to which it is sensible to enforce security goals. The more strictly the principle of least privilege is enforced, the more unwieldy the rules and mechanisms become. We need to balance the scope of technical enforcement mechanisms against the usability and efficiency of the system.

We should note also that we may need to allow some flexibility in the rules in order to be able to deal effectively with exceptional circumstances. It is difficult to foresee all possible scenarios and hence difficult to formulate rules that will deal with all circumstances. We may decide therefore that certain rules, even though they may be theoretically enforceable, are better not enforced and just monitored. To take an example: we may have a rule stating that patient records may only be viewed by the patient's registered doctor or other clinicians with the informed consent of the patient. In exceptional circumstances, e.g., the patient's doctor is unavailable and the patient is in a coma, it may be life-threatening to enforce this rule. It is better to allow this rule to be over-ridden, with suitable warnings and subsequent audit.

Where it is not possible or feasible to technically enforce the policy we must impose social rules (prohibitions, duties and obligations) on the users and deploy social enforcement mechanisms: accountability, responsibility, etc., to coerce the human users into obeying these social rules. Thus, spot checks might be made on patient records to try to spot anomalies. Actions that violate a rule might not be blocked but might be flagged with a warning, logged and subsequently checked.

3.2.3 Faults in the implementation of the technical rules

Let us suppose now that we have found an adequate formulation of the security rules, in the sense of the discussion above. We must now consider the possibility of faults in the implementation of these (abstractly specified) rules that could result in users being able to acquire privileges or accesses to which they are not entitled.

This can happen in a number of ways:

- Simple coding errors. The implemented rules do not match the abstractly specified rules.
- Failure to guarantee the integrity of the rules. For example if there is some way to corrupt the access control matrix or access control lists.
- Failure to properly maintain the rules.
- Errors in mapping the rules onto the architecture, e.g., inadequate scope of the access control mechanisms, for example if the security kernel is by-passable.

This last problem may ultimately be traceable back to flaws in the modelling and verification process: overlooking or inaccurately modelling certain channels and access modes in the architecture for example. Covert channels provide an example this.

3.2.4 Faults in the lower level technical mechanisms

The mechanisms for enforcing the security rules depend on services provided by lower level security mechanisms: authentication, encryption, and so on. Thus our access control rules might be perfectly formulated, implemented and maintained and yet, if the authentication

mechanism fails, an intruder may be able to masquerade as a legitimate user and so usurp their privileges. Similarly, faults in the cryptographic primitives may lead to faults in the authentication protocols, and so forth.

3.2.5 Faults caused by deficiencies of formal models

Most faults discussed in the previous paragraphs can be traced back to an inadequate, or incomplete design. One important reason that such faults are not avoided or removed during system development is the fact that the mathematical models that we construct to perform analysis are, necessarily, abstractions of reality. Despite our best efforts to make the models faithful to the security relevant aspects of reality, there is always the danger of overlooking something significant. Thus, models that abstract from time or from power consumption will not identify attacks based on the adversary's ability to observe these factors and draw inferences from them.

This is a particularly virulent problem when trying to model information flows: any observable may potentially be a source of flows. The choice of model of computation and interaction is thus crucial. Even if we have correctly identified all possible variables we may still have problems arising from non-determinism. If the sensitive activity can have any influence on the resolution of non-determinism visible to the adversary this will give rise to a potential information flow. It is extremely difficult to characterise the assertion that no such influence is possible.

Closely related to this are the assumptions embedded in the architecture, i.e., assumptions about what are the channels, interfaces, etc., of the system. If we overlook channels or incorrectly model the flows (overlook covert channels for example) then our system may harbour unknown information leaks.

3.2.6 Faults in the socio-technical mechanisms

Faults may also be present in the socio-technical mechanisms. Indeed, it may be argued that most system security failures are due, at least in part, to human factors. Social engineering attacks in which users are induced to reveal their passwords to an intruder provide prime examples of this. Writing passwords on post-it notes stuck to the monitor is a classic security failure. System support staff may fail to install up to date security patches.

Socio-technical rules and mechanisms should serve to constrain the behaviour of the human users of the system to prevent such failures: careful selection of staff, good security training and awareness programmes, effective audit trails and auditing procedures and so on. Even so, such mechanisms may be inadequate or simply fail:

- Duties and obligations may be incorrectly or inadequately formulated,
- Users may fail to fulfil their duties and obligations.

3.3 Fault model

In this section, we progressively define a fault model that is appropriate for reasoning about prevention and tolerance mechanisms aimed at ensuring system security. We first revisit the notions of fault, error and failure introduced in Section 2.3 and then elaborate on potential causes of security failures.

3.3.1 Causal chain of impairments

In Section 2.3, the notions of fault, error and failure were defined in terms of a causal chain:

- **fault**: adjudged or hypothesised cause of an *error*;
- **error**: that part of the system state that may lead to *failure*;

- **failure**: delivered service deviates from implementing the system function;

i.e., an error is the manifestation of a fault on the system state (where "state" is taken in a broad behavioural sense) and a failure is the manifestation of an error on the service delivered to the system user.

From an intrusion-detection/tolerance viewpoint, the need for three types of causally-related impairments can be justified by the following remarks:

- It is necessary to distinguish the internal detectable impairment (*error*) from the causing impairment (*fault*) since there may be multiple causes (e.g., intentionally malicious faults vs. accidental faults, cf. Section 2.3) that could give rise to the same detectable impairment.
- It is necessary to distinguish the internal detectable impairment (*error*) from the external impairment (i.e., failure in the service delivered to a user) that intrusion-tolerance techniques aim to prevent. The alternative viewpoint, in which any detectable impairment is deemed to make the system "insecure" in some sense, would make intrusion-*tolerance* an unattainable objective.

Due to the recursive definition of systems in terms of components, a failure at a given level of decomposition may naturally be interpreted as a fault at the next upper level of decomposition, thus leading to a hierarchical causal chain, as illustrated in Figure 7, where the dotted lines represent a "system boundary", at the considered level of decomposition or abstraction.

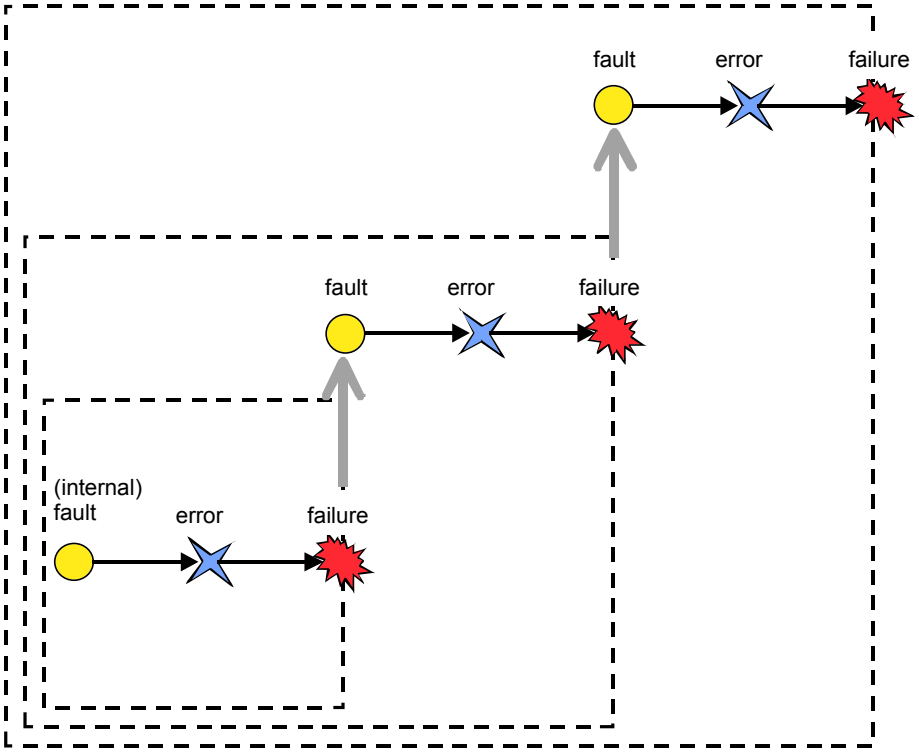


Figure 7 — Hierarchical causal chain of impairments

Ideally, the MAFTIA fault model should enable such a hierarchical interpretation.

3.3.2 Intrusion, attack and vulnerability

An intrusion is defined on Figure 4, page 7, to be a deliberately-malicious software-domain operational fault that originates externally to the (technical) system boundaries. Etymologically, the word "intrusion" comes from the Latin *intrudere* (to thrust in) but current

usage covers both senses of "illegal penetration" and "unwelcome act". Even a malicious interaction fault perpetrated by an *insider* can thus be classed as an intrusion since the intent is to carry out an operation on some resource that is unwanted by the owner of that resource.

A possible alternative to "intrusion" would be the word "attack". However, it would seem that both terms are necessary, but for different concepts. A system can be attacked (either from the outside or the inside) without any degree of success. In this case, the attack exists, but the mechanisms that protect the system or resource targeted by the attack are sufficiently efficacious to *prevent* intrusion. An attack is thus an *intrusion attempt* and an intrusion results from an attack that has been (at least partially) successful.

In fact, there are two underlying causes of any intrusion (Figure 8):

1. A malicious act or *attack* that attempts to exploit a weakness in the system,
2. At least one weakness, flaw or *vulnerability*.

Note that vulnerabilities may be introduced during development of the system, or during operation. Furthermore, such faults can be introduced accidentally or deliberately, with or without malicious intent.

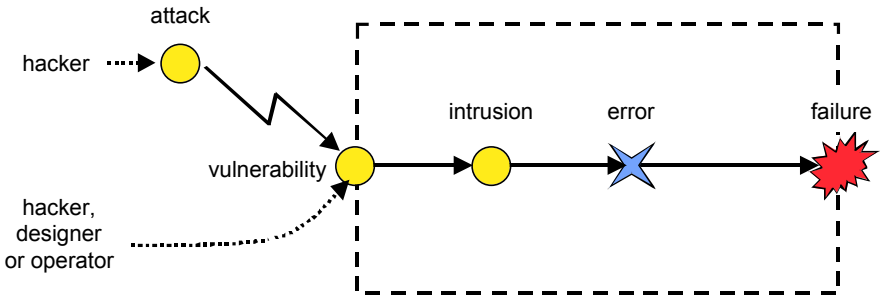


Figure 8 — Intrusion as a composite fault

This is a similar situation to that of externally-induced physical faults: a heavy ion approaching the system from outside is like an attack. The aim of shielding is to prevent the heavy ion from penetrating the system. If the shielding is insufficient, a fault will occur (e.g., a latch-up). Mechanisms can be implemented inside the system to tolerate such externally-induced faults. Since we are essentially concerned with techniques aimed at providing security guarantees in spite of imperfect "shielding" of the considered system, we will later refer to such techniques as *intrusion-tolerance* techniques, which aim to tolerate the fact that vulnerabilities have been successfully exploited by an attacker (who is, *ipso facto*, an *intruder*).

Typical examples of intrusions interpreted in terms of vulnerabilities and attacks are:

1. An outsider penetrating a system by guessing a user password: the vulnerability lies in the configuration of the system, with a poor choice of password (too short, or susceptible to a dictionary attack).
2. An insider abusing his privilege (i.e., a misfeasance): the vulnerability lies in the specification or the design of the (socio-technical) system (violation of the principle of least privilege, inadequate vetting of key personnel).
3. An outsider using "social engineering", e.g., bribery, to cause an insider to carry out a misfeasance on his behalf: the vulnerability is the presence of a bribable insider, which in turn is due to inadequate design of the (socio-technical) system (inadequate vetting of key personnel).

4. A denial-of-service attack by request overload (e.g., the February 2000 DDoS[4] attacks of Web sites): the vulnerability lies partly in the very requirements of the system since it is contradictory to require a system to be completely open to all well-intended users and closed to malicious users. This particular type of attack also exploits design or configuration faults in the many Internet-connected hosts that were penetrated to insert the zombie daemons required to mount a coordinated distributed attack [Garber 2000]. A third vulnerability, which prevents effective countermeasures from being launched, resides in a design fault on the part of Internet service providers not implementing ingress/egress filtering (which would enable the originating IP source address to be traced).

From the above, it is therefore clear that, according to the adopted viewpoint, at least three fault types must be taken into account when reasoning about possible causes of errors that could lead to a security failure:

attack – a malicious interaction *fault,* through which an attacker aims to deliberately violate one or more security properties; an *intrusion* attempt.

vulnerability – a fault created during development of the system, or during operation, that could be exploited to create an *intrusion*.

intrusion – a *malicious*, externally-induced *fault* resulting from an *attack* that has been successful in exploiting a *vulnerability.*

Vulnerabilities are the primordial faults existing inside the components, in particular design or configuration faults (e.g., coding faults allowing program stack overflow, files with root setuid in Unix, naïve passwords, unprotected TCP/IP ports). Note, however, that a successful attacker might purposely introduce a vulnerability (in the form of malicious logic, see below) as a step in his overall plan of attack.

Attacks may be viewed either at the level of human activity of the attacker, or at the level of the resulting technical activity that is observable within the considered computer system.

attack (*human*) – a malicious human interaction *fault* whereby an attacker aims to deliberately violate one or more security properties.

attack (*technical*) – a malicious technical interaction *fault* aiming to exploit a *vulnerability* as a step towards achieving the final aim of the attacker.

Attacks (in the technical sense) are malicious interaction faults that attempt to activate one or more vulnerabilities (e.g., port scans, email viruses, malicious Java applets or ActiveX controls). An attack that successfully activates a vulnerability causes an intrusion. This further step towards failure is normally characterised by an erroneous state in the system that may take several forms (e.g., an unauthorised privileged account with telnet access, a system file with undue access permissions for the attacker). Such erroneous states may be corrected or masked by intrusion tolerance (see Chapter 4) but if nothing is done to process the errors resulting from the intrusion, failure of one or more security properties will probably occur.

We will only qualify the human/technical nature of attacks when necessary; in the absence of qualification, we consider "attack" in its technical sense. *Attacker* is always taken in its human sense, i.e., the malicious person or organization at the origin of attacks. When a technical attack is perpetrated on behalf of the attacker by some piece of code (including malicious logic as discussed in Section 3.3.3 below), we refer to such code as an *attack agent*.

[4] DDoS: Distributed Denial of Service.

3.3.3 Malicious logic

Malicious logic refers to internal, deliberately malicious faults. Such faults may be introduced either during system development, or while the system is in operation [Landwehr et al. 1994]. These faults include the following:

logic bomb – *malicious logic* that remains dormant in the host system till a certain time or an event occurs, or certain conditions are met, and then deletes files, slows down or crashes the host system, etc.

zombie – *malicious logic* that can be triggered by an attacker in order to mount a coordinated attack.

Trojan horse – *malicious logic* performing, or able to perform, an illegitimate action while giving the impression of being legitimate; the illegitimate action can be the disclosure or modification of information (attack against confidentiality or integrity) or a *logic bomb*.

trapdoor – *malicious logic* that provides a means of circumventing access control mechanisms.

virus – *malicious logic* that replicates itself and joins another program (system or application) when it is executed, thereby turning into a *Trojan horse*; a virus can carry a *logic bomb*.

worm – *malicious logic* that replicates itself and propagates without the users being aware of it; a worm can also carry a *logic bomb*.

Monitoring programs, such as network sniffers, can also be considered as malicious logic when they have been inserted illegally.

In the definition of Trojan horse is hidden another notion concerning the *mode* of action of the malicious logic. Often the Trojan horse is a sort of "subversive agent" for attacking the confidentiality or integrity of its host (note that an illegally installed sniffer is a similar sort of subversive agent). However, the important concept behind "Trojan horse" is that of being hidden in a program that the unsuspecting user believes to be legitimate.

Malicious logic can be divided into intentionally installed *vulnerabilities* (e.g., a trapdoor) and *attack agents*. The latter might be classified according to the following dimensions:

- **propagation**: self-propagating (i.e., as in *virus* or *worm*); no propagation (one-off result of an intrusion);
- **trigger conditions**: continuously activated (e.g., *Trojan horse*); serendipitous activation by unsuspecting victim (e.g., *Trojan horse*); other conditions (specific time, input value, etc.) (i.e., a *bomb* or a *zombie*);
- **target of attack**: local (e.g., a *bomb* or a *Trojan horse*) or distant (i.e., a *zombie*);
- **aim of attack**: disclosure (confidentiality); alteration (integrity), denial of service (availability).

3.3.4 Intrusion propagation

Let us now return to the notion of a hierarchical causal chain of impairments as represented by Figure 7, page 30. A security failure at one level of decomposition of the system may be interpreted as an intrusion at the next upper level. For example, the failure of an authentication and authorisation mechanism to prevent system penetration by a malicious user is clearly an intrusion as seen from the containing system, in this case, the system to which access is being mediated (Figure 9). The containing system might now detect and recover from any resulting errors (e.g., abnormal behaviour), and thereby prevent a security failure at its level. If it is unsuccessful in this, then the next upper containing system may view the

lower-level security failure as an intrusion, and so on. Another example is a buffer overflow in a program: at the second level, the operating system may or may not prevent its own failure (depending on what rights the failed program has), and at the third level, a distributed system may or may not be able to tolerate the failure of an entire node.

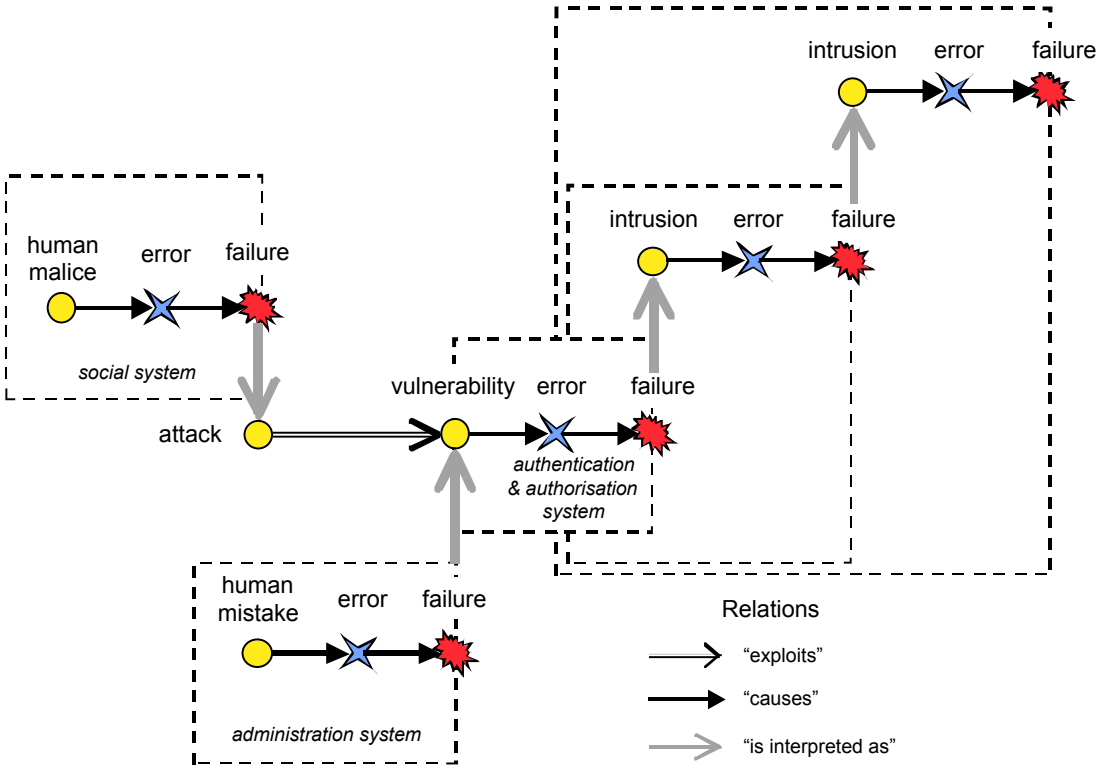


Figure 9 — Attack, vulnerability and intrusion in a hierarchical causal chain

Moreover, depending on the adopted viewpoint at a given level, the intrusion may also be viewed as an attack, as the installation of a vulnerability, or as an attack agent. Indeed, the intrusion may manifest itself as a further attack (the attacker directly exploits his successful attack in order to proceed towards his final goal); by the creation of new vulnerabilities (e.g., a system file with undue access permissions for the attacker, or malicious logic creating a *trapdoor*) or by the insertion of malicious logic that can act as an agent for the attacker sometime in the future (e.g., a *zombie*).

Figure 9 also traces back the causal chain through the failures of two other "systems":

- The vulnerability in the authentication system is due to the failure of the administration system to prevent the administrator from creating the vulnerability.
- The attack that exploited the vulnerability in the authentication system is due to the failure of the social system to deter the attacker from attacking.

3.3.5 Theft and abuse of privilege

In Section 3.3.2, we referred loosely to attackers as being either "outsiders" or "insiders". What exactly is the distinction between the two?

In common parlance, an insider is "a person within a society, organisation, etc. or a person privy to a secret, especially when using it to gain advantage" [OMED 1992].

The first part of this definition can be interpreted in terms of the *rights* of the considered person. A person has a *right* on a specified object within the system if and only if he is authorised to perform a specified operation on that object — a right is thus an object-

operation pair. The set of rights of the considered person is that person's *privilege*. An *outsider* might thus be defined as a person who has no privilege, i.e., no rights on any object in the system. Inversely, an *insider* is thus any individual who has some privilege, i.e., some rights on objects in the system.

Consider now the case of an "open" system, such as a public web server. Such systems grant to all users at least read access rights on certain objects within the system so, with the above definitions, all users would be considered as insiders. The very notion of an outsider, as defined above, is only relevant for *closed* systems.

An alternative distinction is thus necessary for open systems. The second part of the dictionary definition of an insider relates both to the *knowledge* of the considered person and the *illegal use* of this knowledge[5]. The distinction between outsider and insider must thus be made in terms of the types of intrusion that can be perpetrated by the considered person:

theft of privilege: an unauthorised increase in privilege, i.e., a change in the privilege of a user that is not permitted by the system's access control policy.

abuse of privilege: a misfeasance, i.e., an improper use of authorised operations.

These two notions are illustrated by Figure 10, which considers a subset of the universe of object-operation pairs of the considered system, rather than the complete system, and the current privileges of two users (*a* and *b*).

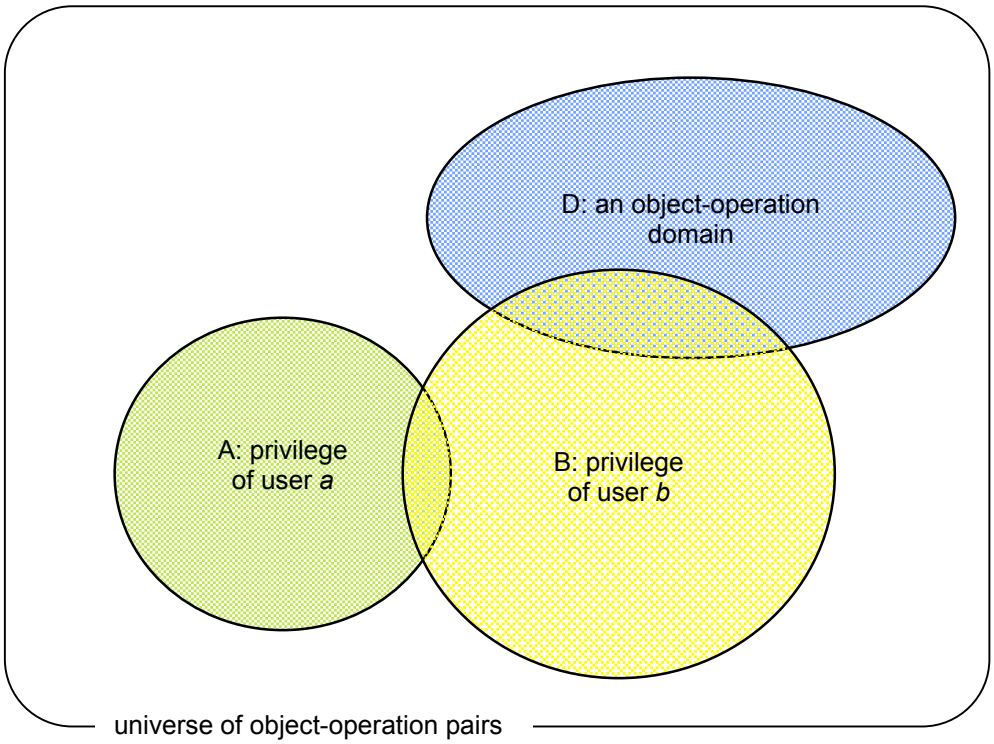


Figure 10 — Outsider (user a) vs. insider (user b) with respect to domain D

In Figure 10, user *a* is currently an outsider with respect to a given domain D of object-operation pairs since his privilege does not intersect that domain. User *a* can only act within domain D by stealing a privilege beyond his current privilege. User *b* is an insider with respect to domain D but an outsider with respect to sub-domain $D - B$. User *b* can thus perpetrate both sorts of intrusion on domain D: an abuse of privilege within $D \cap B$ or a theft

[5] The relationship between *knowledge* and *right* needs to be explored further, especially in terms of concepts such as the *need to know* and the *principle of least privilege*.

of privilege within $D - B$. With respect to a given domain of object-operation pairs, we can thus define outsider and insider as follows:

outsider: a human user not authorised to perform any of a set of specified operations on a set of specified objects, i.e., a user whose (current) privilege does not intersect the considered domain of object-operation pairs.

insider: a human user authorised to perform some of a set of specified operations on a set of specified objects, i.e., a user whose (current) privilege intersects the considered domain of object-operation pairs.

3.3.6 Intrusion containment regions

In this section, we introduce the notion of an intrusion containment region by analogy with the notion of a fault containment region, which has proven useful as a concept when tolerating accidental faults (see, for example, [Smith 1986]).

A fault containment region (FCR) can be defined as: a set of components that is considered to fail (a) as an atomic unit, and (b) in a statistically independent way with respect to other such FCRs. Then, with the following assumptions:

A1: the behaviour of a faulty FCR is unrestricted[6];

A2: there are a bounded number of faulty FCRs in the considered fault-tolerant system;

it is possible to define formal fault-tolerance properties (e.g., agreement) for the fault-free FCRs, but faulty FCRs are disregarded since, according to assumption A1, their behaviour is unrestricted.

When defining the correctness of a mechanism designed to tolerate *intrusions*, a similar restriction to fault-free components must apply since no assumptions can be made about what an intruder or a corrupted component can or cannot do.

An intrusion-tolerant system is aimed at guaranteeing certain security properties, despite the fact that some components of the system might be compromised, by either corrupt system administrators or corrupt users. Consider now that users (and administrators) of the considered computer system access the latter by means of an "access point", i.e., a terminal or a workstation (Figure 11).

[6] In practice, there is always *some* assumption about what a faulty FCR is not allowed to do in the sense that it should not be able to change the *structure* of the considered fault-tolerant system. For example, in Byzantine agreement, a disloyal general is only allowed to change messages (in arbitrary ways), but is not allowed to kill his colleagues, or to create clones of himself to falsify the majority.

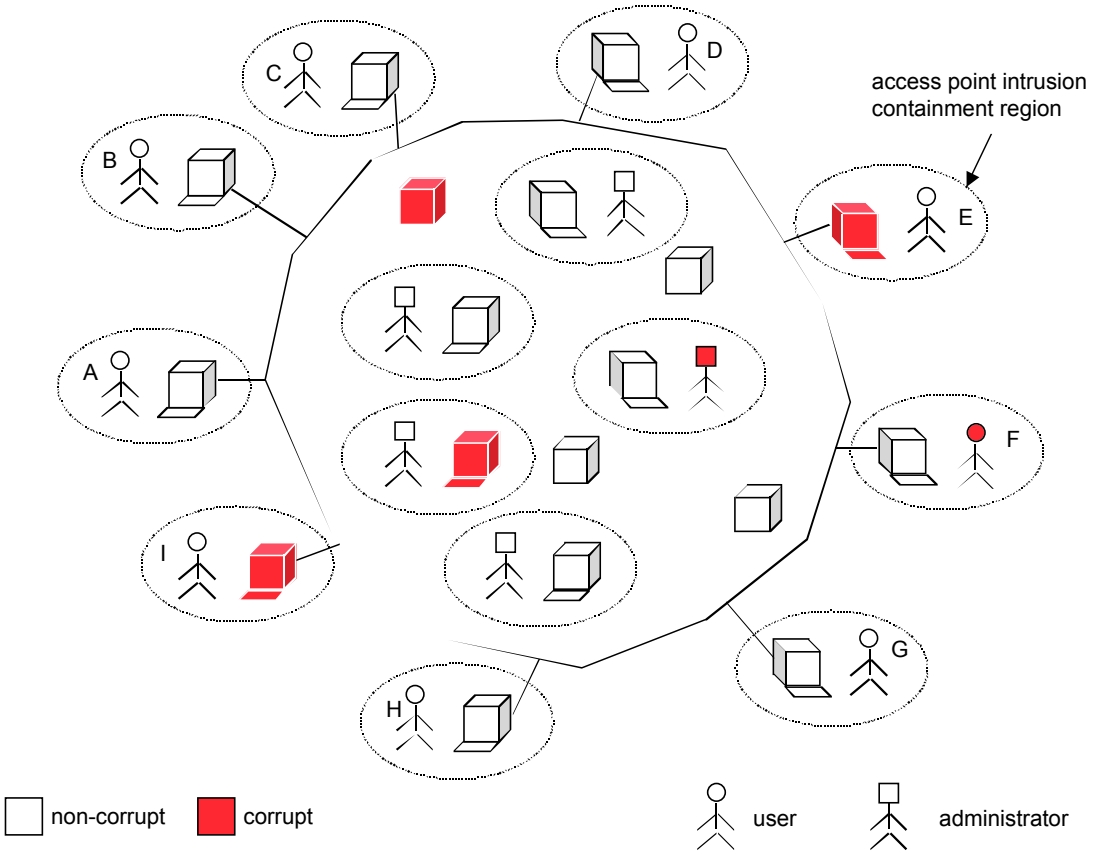


Figure 11 — Corrupt vs. non-corrupt access points and users

If an access point has been corrupted (e.g., by a Trojan horse that logs or modifies confidential inputs or outputs) then it is clear that the user of that access point cannot be given any security guarantees (case of users E and I in Figure 11).[7] Also, it is of no interest to give a security guarantee to a corrupt user, even if his access point is non-corrupt (case of user F in Figure 11). Indeed, from the security viewpoint, a user (or an administrator) and his corresponding access point constitute a single "intrusion containment region": it is of no import to the rest of the system whether a user or his access point is corrupt.

Consequently, it is clear that security guarantees can be given only with respect to a set of non-corrupt access point intrusion containment regions, e.g., access points A, B, C, D, G and H in Figure 11. For non-corrupt users of non-corrupt access points, an intrusion-tolerant system should be able to provide guarantees about the confidentiality, integrity and availability of the data owned (or, equivalently, the service purchased) by those users, despite the fact that there are (a certain number of) corrupt components, administrators or users of the system. A similar concept is introduced in [Pfitzmann & Waidner 1994], where security properties are specified in terms of a subset of access points that together constitute the interface to the concerned parties, i.e., those parties who mutually trust each other but distrust other parties (other users, access points or system components).

It might be possible to generalise this notion of an intrusion containment domain beyond that of just access points. Indeed, an intrusion containment domain may be interpreted in terms of the set of rights that an intruder has managed to obtain, i.e., his current privilege domain (cf. Section 3.3.3). The intruder's current privilege domain defines the extent of control that he has over the system. The intruder may maliciously misuse any object-operation pair within

[7] The protection of an access point against intrusions should thus be under the responsibility of the corresponding user: a reckless user cannot be given any security guarantees.

that privilege domain, so it would be wise to consider the whole domain as corrupt, *if* of course one were able to dynamically infer what constitutes that domain at a given instant.

A specific case may be the "uses" relation within operating systems, which would lead to a general directed graph of dependencies between components, generalising the equivalence relation leading to FCRs. For example, all servers using an operating system with a security kernel fail if the kernel fails, but not vice versa, and a user program fails if a server that it uses fails, but not vice versa. While cryptographic and distributed-system measures often work with a model of intrusion containment regions, a general directed graph may be more suited to modelling typical "intrusion-detection" measures.

3.4 Security methods

Equating *attack* (in both the human and technical senses), *vulnerability* and *intrusion* with fault, and applying the definitions given in Section 2.5, we can obtain *a priori* sixteen methods for ensuring or assessing security (Table 8). However, not all of these sixteen methods are distinguishable.

We in fact obtain ten distinguishable methods, which are presented in the following subsections.

Table 8 — Classification of security methods

	Attack (human sense)	**Attack** (technical sense)	**Vulnerability**	**Intrusion**
Prevention (how to prevent occurrence or introduction of…)	deterrence, laws, social pressure, secret service…	firewalls, authentication, authorisation…	semi-formal and formal specification, rigorous design and management…	= attack & vulnerability prevention & removal
Tolerance (how to deliver correct service in the presence of…)	= vulnerability prevention & removal, intrusion tolerance		= attack prevention & removal, intrusion tolerance	error detection & recovery, fault masking, intrusion detection, fault handling
Removal (how to reduce number or severity of…)	physical countermeasures, capture of attacker	preventive & corrective maintenance aimed at removal of attack agents (i.e., some forms of malicious logic)	1. formal proof, model-checking, inspection, test… 2. preventive & corrective maintenance, including security patches	⊆ attack & vulnerability removal
Forecasting (how to estimate present number, future incidence, likely consequences of…)	intelligence gathering, threat assessment…	assessment of presence of latent attack agents, potential consequences of their activation	assessment of: presence of vulnerabilities, exploitation difficulty, potential consequences…	= vulnerability & attack forecasting

3.4.1 Fault prevention

In Section 2.5, fault prevention is defined as "how to prevent the occurrence or introduction of faults". Equating *attack*, *vulnerability* and *intrusion* with fault, we obtain three clearly distinguishable sets of fault-prevention methods:

attack prevention (*human sense*): how to prevent the occurrence of human attacks;

With attack taken in the human sense, this includes deterrence measures such as social pressure, laws and their enforcement.

attack prevention (*technical sense*): how to prevent the occurrence of technical attacks;

When attack is taken in its technical sense, attack prevention consists of the introduction of mechanisms such as authentication, authorisation and firewalls, which "prevent" attacks in that they "push back" the attacks to the level of the additional barriers these mechanisms introduce.

vulnerability prevention: how to prevent the occurrence or introduction of vulnerabilities;

This includes measures going from semi-formal and formal specification, rigorous design and system management procedures, up to and including user education (e.g., choice of passwords).

Note that *intrusion prevention* (as opposed to intrusion *tolerance*) can be seen as the combined application of attack and vulnerability prevention, as well as attack and vulnerability removal, i.e., the classic security techniques.

3.4.2 Fault tolerance

In Section 2.5, fault tolerance is defined as "how to deliver correct service in the presence of faults". Equating *attack*, *vulnerability* and *intrusion* with fault does not lead to clearly distinguishable sets of methods. First, since an intrusion cannot occur in the absence of vulnerability, intrusion tolerance and vulnerability tolerance are equivalent in the sense that tolerance of an intrusion implies tolerance of the vulnerability or vulnerabilities that were exploited to perpetrate the intrusion. To conform to current usage, we will refer to *intrusion tolerance*. Note that the presence of vulnerabilities can be tolerated if no attacks occur, so attack prevention and removal are also a form of vulnerability tolerance.

Similarly, attack tolerance does not define a separate set of methods beyond vulnerability prevention and removal, and intrusion tolerance. Hence, we obtain one distinguishable set of fault-tolerance methods:

intrusion tolerance: how to provide correct service in the presence of intrusions;

Admitting that attack, vulnerability and intrusion prevention measures are always imperfect, intrusion tolerance aims to ensure that the considered system provides security guarantees in spite of partially successful attacks. Techniques for achieving intrusion tolerance will be addressed in Chapter 4.

3.4.3 Fault removal

In Section 2.5, fault removal is defined as "how to reduce the number or severity of faults". Fault prevention and fault removal are sometimes grouped together under the term "fault avoidance". Fault removal may occur either before or after a system is put into operation. In the latter case, it is called maintenance (note that the factor that distinguishes maintenance from fault tolerance is that the former requires participation of an external agent).

Equating *attack*, *vulnerability* and *intrusion* with fault leads to the following interpretations of fault removal.

Attack removal can be considered during system operation, for both human and technical attacks, through the notion of countermeasures. In both cases, it is not so much the attack that is removed, but the entity perpetrating the attack, i.e., the (human) attacker or the (technical) attack agent.

For vulnerabilities, removal consists of verification procedures aimed at identifying and removing flaws that could be exploited by an attacker, including maintenance procedures

aimed at removing vulnerabilities identified during system operation (corrective maintenance) or in similar systems (preventive maintenance).

We have no meaningful separable interpretation of fault removal in terms of intrusions other than preventive and corrective maintenance procedures aimed at removing attack agents and vulnerabilities resulting from intrusions.

Hence, we obtain three distinguishable sets of fault-removal methods:

attack removal (*human sense*): how to reduce the number or severity of human attacks;

This covers human countermeasure techniques aimed directly against the attacker

attack removal (*technical sense*): how to reduce the number or severity of technical attacks;

This covers maintenance actions aimed at removing malicious logic acting, or capable of acting, as attack agents.

vulnerability removal: how to reduce the number or severity of vulnerabilities;

During system development, this covers verification procedures such as formal proof, model-checking and testing, specifically aimed at identifying flaws that could be exploited by an attacker. Identified flaws may then be removed by correcting the code.

During system operation, vulnerability removal corresponds to preventive and corrective maintenance actions such as applying a security patch, withdrawing a given service, changing a password, removal of malicious logic implementing a trapdoor, etc.

3.4.4 Fault forecasting

In Section 2.5, fault forecasting is defined as "how to estimate the present number, the future incidence, and the likely consequences of faults". Equating *attack*, *vulnerability* and *intrusion* with fault, we obtain three clearly distinguishable sets of fault-forecasting methods:

attack forecasting (human sense): how to estimate the present number, the future incidence and the likely consequences of (human) attacks.

This includes intelligence gathering, threat assessment and attack warning

attack forecasting (technical sense): how to estimate the present number, the future incidence and the likely consequences of (technical) attacks.

This corresponds to an assessment of the present number of latent attack agents, and the future incidence and the likely consequences of their activation.

vulnerability forecasting: how to estimate the present number, the future incidence and the likely consequences of vulnerabilities.

This includes the gathering of statistics about the current state of knowledge regarding system flaws, and the difficulties that an attacker would have to overcome in order to take advantage of them.

Security risk analysis can be viewed as a combination of all three forecasting methods.

Finally, note that the assessment of the effectiveness of intrusion-detection mechanisms also falls into the category of fault forecasting methods (similarly to coverage assessment in traditional-fault tolerance). However, the "faults" whose incidence is being forecasted are the design faults in such detection mechanisms rather than the intrusions they aim to detect.

Chapter 4 Intrusion tolerance

This chapter focuses on one of the eight security methods identified in Section 3.4, namely *intrusion tolerance*, defined as "how to provide a service capable of implementing the system function despite intrusions" and aimed at ensuring that a system provides guarantees of security despite partially successful attacks.

Before doing so, however, Section 4.1 first discusses what is meant by *intrusion detection*. In Section 4.2, we give a model for describing intrusion-detection systems. Then, in Section 4.3, we discuss intrusion tolerance in the light of the core dependability definitions relative to fault tolerance given in Section 2.5.2. Finally, in Section 4.4, we define a general framework that integrates the notions of intrusion detection and intrusion tolerance.

4.1 Intrusion detection

In Section 2.3, a *fault* is defined to be the adjudged or hypothesised cause of an *error*, the latter being that part of the system state that *may cause a subsequent failure*. A security failure will occur if a system fails to deliver a secure service, and the role of an intrusion detection system is to detect errors before they lead to security failures. But what is the nature of a security failure?

Section 3.1 contains a discussion of the role of a security policy in defining security requirements. We consider a security policy to be made up of goals and rules. The goals define the security requirements of the system, and thus a violation of the goals constitutes a security failure. The security rules serve to support the security goals. A violation of a security rule might not of itself constitute a security failure, but will typically put the system in a state that is deemed to be more liable to failure, and thus corresponds to an error.

The goals of a security policy are stated in terms of the basic security properties of confidentiality, integrity and availability, and thus, the definition of security failure is naturally derived from loss of confidentiality, integrity or availability (as defined by the goals of the considered security policy). However, there is currently no agreed definition of what might constitute an *error* from the security viewpoint, although we consider the violation of a security rule to result in an erroneous state, as discussed above. Despite this lack of consensus, current literature refers to "intrusion detection" which, from the dependability concept viewpoint, might lead one to equate intrusion with "error", rather than "fault"[8]. In reality, current literature uses the term "intrusion detection" to cover a *spectrum* of techniques. To paraphrase [Halme & Bauer]: "Intrusion detection may be accomplished:

- after the fact (as in post-mortem audit analysis),
- in near real-time (supporting SSO[9] intervention or interaction with the intruder, such as a network trace-back to point of origin), or
- in real time (in support of automated countermeasures)."

---

[8] Note, however, that it is also quite common in the literature on tolerance of physical faults to find the term "fault detection" used on one of two ways:

a) As a clumsy synonym for "error detection" (since detection of an error implies, rather indirectly and perhaps falsely, the "detection" of its cause)

b) As the designation of a mechanism that seeks out (dormant) faults by running a test procedure to activate them as errors that can be detected by an error detection mechanism.

[9] SSO: System Security Officer.

From the dependability concept viewpoint, these three types of intrusion detection can be interpreted respectively as:

- off-line fault diagnosis (as part of curative maintenance);
- error detection and on-line fault diagnosis (to an operator-assisted fault handling facility);
- error detection (as a preliminary to automatic error recovery), or error detection and on-line fault diagnosis (as a preliminary to automatic fault handling).

Further confusion is introduced by the opposition in [Halme & Bauer] between a "manually reviewed IDS"[10] (called a passive IDS in [Debar et al. 1999]), and "Intrusion Countermeasure Equipment (ICE)" and an "autonomously acting IDS" (sic) (called an active IDS in [Debar et al. 1999]), which clearly go beyond just detection. The notion that an IDS might include more than just detection, but also the actions triggered by detection, also appears in the Common Intrusion Detection Framework (CIDF) [Porras et al.]. This framework, which we will re-visit later in this chapter, defines the notion of "response units", that take inputs from other CIDF components to carry out "some kind of action … [on their behalf, including] … such things as killing processes, resetting connections, altering file permissions, etc.".

Here, we will prefer to make a distinction between detection *per se* and response, be it manual or automatic. This concurs with the definition given in [NSA 1998], where intrusion detection is defined as: "Pertaining to techniques which attempt to detect intrusion into a computer or network by observation of actions, security logs, or audit data. Detection of break-ins or attempts either manually or via software expert systems that operate on logs or other information available on the network." This is also in line with the charter of the Intrusion Detection Working Group (IDWG) of the Internet Engineering Task Force (IETF) [IETF], which speaks of "intrusion detection *and* response systems".

To conform to the spirit of the NSA definition above, we will avoid using "intrusion detection" in the limited sense of "error detection" but extend it to include some degree of fault diagnosis. To this end, we adopt the following definitions:

intrusion detection: concerns the set of practices and mechanisms used towards detecting errors that may lead to security failure, and/or diagnosing attacks.

intrusion-detection system: an implementation of the practices and mechanisms of intrusion detection.

Our definition of intrusion detection draws attention to the fact that we are particularly interested in detecting errors that may lead to security failure since the ultimate aim of such a system is to provide inputs:

- to a system administrator (an SSO) who might carry out further diagnosis and initiate litigation and/or appropriate countermeasures to avert security failures, or
- to an automatic countermeasure mechanism to avert security failures, i.e., to *tolerate* intrusions.

However, the definition also covers a second important aim of intrusion detection, that of gathering information about new forms of attack, for which new defences will need to be devised.

4.2 Intrusion-detection model

We present a model of intrusion-detection systems according to function, derived as a refinement of the Common Intrusion Detection Framework (CIDF) [Porras et al.]. When

[10] IDS: Intrusion Detection System.

possible, we use the language of the CIDF although some refinement has been necessary. We additionally address issues of channels between components.

The CIDF classifies components of an intrusion-detection system into four different categories. We recap briefly:

- An **e-box,** or event generator, is a component that gathers event information.
- An **a-box,** or analysis box, analyses event information toward detecting errors and diagnosing faults. The output of an analysis box may provide information to other analysis boxes.
- A **d-box,** or database, provides persistence for the intrusion-detection system. This facility will take on different forms depending upon use. It may be a complex relational database or it may be a simple text file.
- An **r-box,** or response box, is the portion of the system that acts upon the results of analysis. According to [Porras et al.], automated responses may include killing processes, resetting connections, or activating degraded service modes. In line with the discussion in Section 4.1, we do not consider the r-box to be part of intrusion detection *per se*, but as part of the set of facilities providing error recovery, fault isolation and system reconfiguration in a general intrusion-tolerance framework.

Figure 12 presents a refinement of the CIDF model, which explicitly identifies sub-components of the e-box, and the fact that there can be multiple e-, a- and d-boxes.

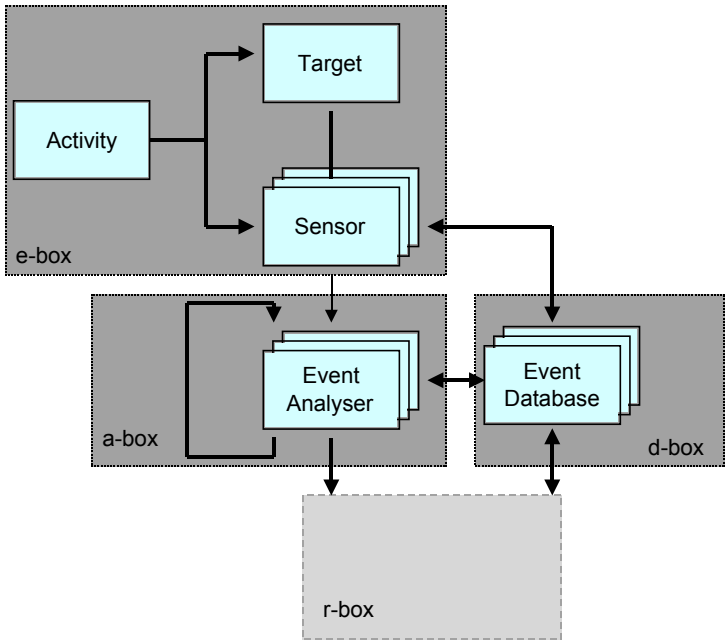


Figure 12 — Intrusion-detection system components

We note that the decomposition may not correspond to particular physical boundaries. Vulnerabilities, and hence targets, exist at several different abstraction and implementation layers so that our model must be applicable at several layers. The boundaries between components are determined by the level of abstraction with which we view the system: people, LANs, machines, processes, memory pages, etc.

The intention is that the several different sensors may generate information stemming from the same root cause, passing it to a cascading array of analysis components in a topologically arbitrary manner (Figure 13).

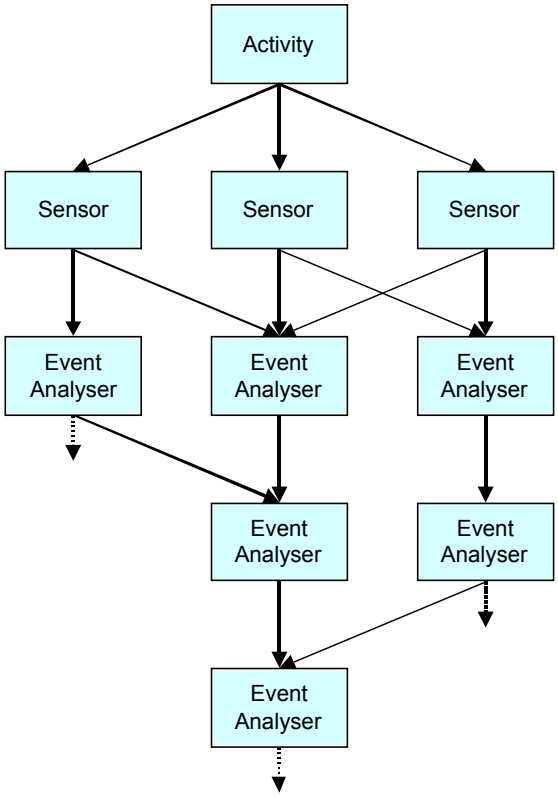


Figure 13 — Cascaded intrusion-detection topology

4.2.1 Event generator

We subdivide what is termed an *e-box* in the language of the CIDF, into three components (*activity, target, sensor*). We have found it necessary to create this subdivision for three reasons [Alessandri 2001]:

- To model an *in vivo* system, we need to consider activity in the system.
- To model the real-world reality of imperfect observation, we need to separate the target of an attack from the sensors used to detect the attack.
- To allow several different sensor boxes for a single target.

The *activity* is the collection of base causes of events in the system. This includes normal user activity, system administration, malicious activity, and spurious events (power failure, system and network crashes, background radiation in the universe, etc.).

The *target* is the component that we are trying to monitor. We assume that the activity has some channel to the target.

The *sensor* is the component of the system collecting raw data (e.g., a sniffer or an audit log). There are two kinds of sensor: sensors that observe the effect of the activity on the target (e.g., host-based sensors), and sensors that seek to observe activity independently of the target (e.g., network-based sensors). Thus, the activity can influence the sensors either directly or indirectly.

The role of the sensor is merely to record raw events (no specifically intrusion-detection logic exists in this component). We note that the sensor may very well be imperfect in the sense that it may not sense all raw events of interest. In certain settings, such as a web daemon recording requests to a (target) script, the sensor is capable of observing all events relative to the target, so it has the possibility of being very reliable (essentially perfect).

Experience has shown that vulnerabilities exist in all places and range through all levels from low-level hardware to high-level social interactions and procedures. Moreover, the exploitation of a vulnerability (i.e., the intrusion) at one level may be concealed using vulnerabilities provided by another.

Naturally enough, different sensors focus on different views of vulnerabilities and their exploitation. Various sensors have different deployment and computational costs and requirements while the different views offer different advantages and possibilities. Optimal deployment is a series of balanced tradeoffs:

- Sensitivity: volume of information vs. analysability
- Deployment: ease vs. completeness
- User rights: privacy vs. visibility

Detection of a violation of the security policy defined at the application level with a network-based sensor would be computationally infeasible. On the other hand, global deployment of application-based sensors may prove too expensive (it is difficult to equip an application with intrusion-detection hooks without the application source code). A network-based IDS on a gigabit per second network link offers an expansive view but will not be able to carry out an in-depth analysis. A host integrity check offers an in-depth but localised view.

Many applications and services that are possible conduits for intrusion offer adequate logging and audit information, either directly or indirectly, to perform intrusion detection. Thus, while it would be unrealistic to expect intrusion-detection logic to be included at all layers, it is still possible to provide intrusion detection for a range of layers.

Ultimately, a complete view of the system is required and all layers of the system must be directly or indirectly visible to the IDS. Some redundant combination of logging, specialised micro-analyser, mid-level and high-level sensors would provide the needed observations.

4.2.2 Event analysis

The *event analysis* boxes successively transform, filter, normalise, and correlate data, adding semantic relevance and reducing volume at each stage. A single event analysis box may take its input from several different producers (both from sensor boxes and other event analysis boxes) and may feed its output to several different consumers in a topologically arbitrary manner (cf. Figure 13).

As with sensors, analysis boxes have differing needs and costs so that deployment is a matter of balanced tradeoffs. A high-level reasoning engine requiring significant computational resources per received event would be quickly overwhelmed by a network scan reported as single events. On the other hand, a high-level reasoning engine may not be able to allot the resources needed to perform subtle correlation.

Further constraints on the distribution and topology of analysis boxes are imposed by the localisation of implementations. An analysis box checking the logs of a web server may be required by practicality to be *near* the web server in terms of management structures while there may also be the need for a global view of web servers for complete analysis.

These constraints are further complicated by the frequent need to combine the observations coming from sources under different management chains.

4.2.3 Event database

An *event database* provides persistence for the IDS. This may be for use in off-line error detection, in intrusion analysis, or as evidence justifying response. This facility has a multi-layered structure similar to that of the entire system. At the lowest level, it may take the form of a simple file. At the highest level, it may be a distributed relational database. We assume

that the database may be interactively queried either by the event analysis boxes or by the response boxes that directly require its contents.

An important aspect of the event database that is *not* addressed in Figure 12 arises from the need to view data with varying degrees of resolution. The use of a single database for an entire enterprise would present serious scalability and privacy issues. The use of multiple databases raises the issues of how to access them and how to meaningfully collate the data obtained from each. This aspect mirrors an identical one for event analysis.

4.2.4 Channels between intrusion-detection components

Channels between components are of course susceptible to failure. They must thus provide integrity (resistance to message alteration and deletion), authenticity (resistance to message insertion), and quality of service (guaranteed delivery or observable failure). Confidentiality features may be required in settings where logging information could prove dangerously useful to an attacker. This includes, for instance, anonymity (e.g., ensure confidentiality of the identity of a person who has root access) and privacy (e.g., ensure confidentiality of personal data).

The mechanisms for such provisions vary with the channels themselves: a TCB (trusted computing base) may offer all of these for IPC (inter-process communication) whereas network connections may need to resort to redundancy and cryptography.

There are several concerns to be addressed:

- An attacker can interrupt the entire channel.
- An attacker can place a smart filter on the channel that hides only the attacker's activities.
- An attacker can interfere with or hijack the entire channel.
- The channel can be eavesdropped upon.

These problems can be addressed in different ways with different costs:

- A heartbeat event ensures that the channel is alive.
- A cryptographic hash chain added to the event stream prevents event deletion.
- Authentication codes prevent event insertion, and event stream hijacking.
- Encryption can prevent the eavesdropping of events.

Such techniques apply not only to transmission but also to storage. Should the logs be stored in a potentially vulnerable location, we can use well-known cryptographic techniques that provide so-called "forward" secrecy [Menezes et al. 1996, Schneier 1996].

4.2.5 Towards an intrusion-tolerant IDS

Chapter 6 of [Dacier 2002] contains a proposal for building an intrusion-tolerant IDS using these mechanisms and techniques, and a prototype demonstrator has been constructed according to these principles. The demonstrator uses redundant sensor networks and event analysers that communicate via secure channels using Byzantine agreement protocols. A heartbeat monitoring mechanism and an immortaliser component provide additional protection against malicious attacks designed to crash particular components of the IDS.

4.3 Interpretation of core fault-tolerance concepts

We now consider intrusions in the broader context of intrusion-*tolerance*. We re-examine the notion of fault-tolerance as defined in the core dependability concepts (Section 2.5.2). Those concepts make a distinction between: (a) *error detection and error handling*, aimed at

preventing errors from leading to (catastrophic) failure, and (b) *fault handling*, aimed at preventing the recurrence of errors.

4.3.1 Error detection

Error detection is a necessary preliminary to achieving rollback or rollforward recovery, or compensation by switchover, but is not strictly necessary if compensation is carried out systematically (i.e., fault masking). However, irrespective of the error handling method employed (if any), error detection is necessary if subsequent fault handling or curative maintenance actions are to be undertaken.

By definition, error-detection (and error handling) techniques need to be applied to all errors irrespectively of the specific faults that caused them. In particular, the malicious or non-malicious nature of faults is not of concern for at least two reasons:

- Determination of whether or not the cause of an error is malicious is not a computational matter (it is a concern rather of psychology).
- We would not want to suppress the notification of a potentially dangerous error merely because the adjudged cause was not deemed malicious (thus not an intrusion).

This does not mean however that the *design* of an error-detection technique is independent of the hypothesised fault model. For example, to detect errors caused by internal physical faults, it suffices to introduce some kind of physically redundant checker hardware (ideally, itself self-checking). Classic examples of such redundancy are: duplication and comparison, parity checking, watchdog timers, etc. The physical redundancy in effect provides an independent reference as to what the behaviour of the monitored hardware should be.

The intrusion-detection community commonly identifies two categories of error-detection techniques that differ according to the type of reference with which the observed system behaviour is compared [Halme & Bauer] (see Figure 14)[11]:

- Anomaly-detection techniques, which compare observed activity against normal usage profiles (in [Debar et al. 1999], these are called behaviour-based methods).
- Misuse-detection techniques, which check for known undesired activity profiles (in [Debar et al. 1999], these are called knowledge-based methods).

Here “anomaly” is definitely being used in the traditional sense of “error”, whereas “misuse” has an element of fault diagnosis since error patterns related to previously identified intrusions are being searched for.

[11] [Halme & Bauer] actually also identifies “hybrid misuse/anomaly detection” and “continuous system health monitoring”. The former is clearly not a separate form of detection and the latter can be viewed as a form of anomaly detection, since it applies to “suspicious changes in system-wide activity measures and system resource usage”.

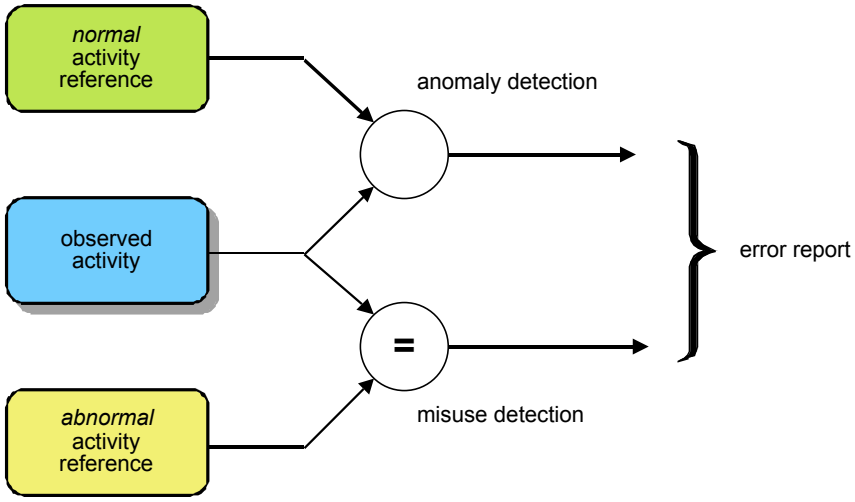


Figure 14 — Detection paradigms

The rules contained within the system's security policy (cf. Section 3.1) provide an important reference regarding what observed activities should be considered as erroneous from a security viewpoint[12]. The rules embodied in the policy may cover both permitted activities (i.e., a normal activity reference) and prohibited activities (i.e., an abnormal activity reference).

Another important error-detection reference, one that is particularly pertinent to MAFTIA, is that provided by subsystems using techniques such as secret-sharing, fragmentation-redundancy-scattering and other trust distribution mechanisms, usually (but not necessarily) implemented with sufficient redundancy to allow intrusion-tolerance through masking (see Section 4.3.2 below). Such techniques provide mutual references of "normal" activity, and should thus be considered as important sources of error-detection reports.

Error-detection techniques are rarely perfect. In traditional fault-tolerance, the degree of perfection of an error-detection mechanism is measured in terms of error-detection coverage (the probability of an error being detected) and latency (the time until an error is detected). Only in rare cases is one interested by the level of "incorrect detections" or "false suspicions" since the monitored component and the error detector are often considered lumped together as a single "self-checking component". However, in intrusion detection, it is necessary to separately consider the various possibilities for incorrect decisions (see Figure 15):

false negative – the *event* corresponding to the incorrect decision not to rate an activity as being erroneous (i.e., no alarm raised due to poor *coverage*, due to either insufficient asymptotic coverage or excessive latency); also called a "miss".

false positive – the *event* corresponding to the incorrect decision to rate an activity as being erroneous; also called a "false alarm".

The corresponding favourable events are:

true negative – the *event* corresponding to the correct decision not to rate an activity as being erroneous.

true positive – the *event* corresponding to the correct decision to rate an activity as being erroneous.

[12] Note, however, that according to our definition (cf. Section 3.1), a security *failure* only occurs when there is a violation of one or more of the security *goals* specified by the security policy.

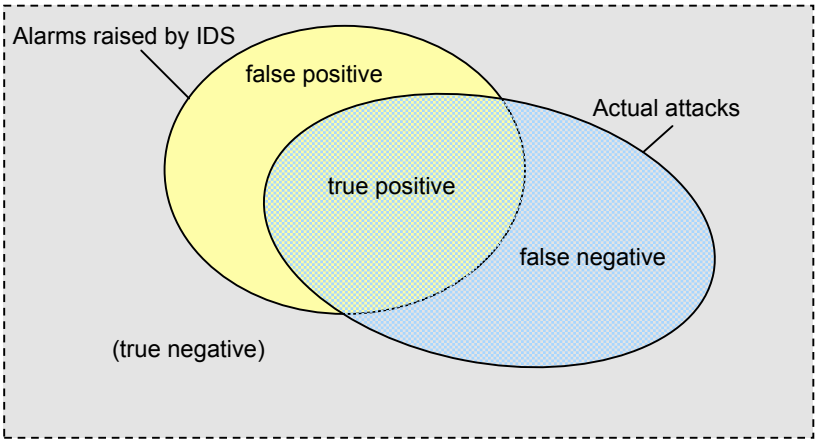


Figure 15 — IDS events

Given that the distributions of normal versus abnormal activity will usually overlap, an IDS needs to be tuned in order to make a compromise between the rates of false negatives and false positives (see Figure 16).

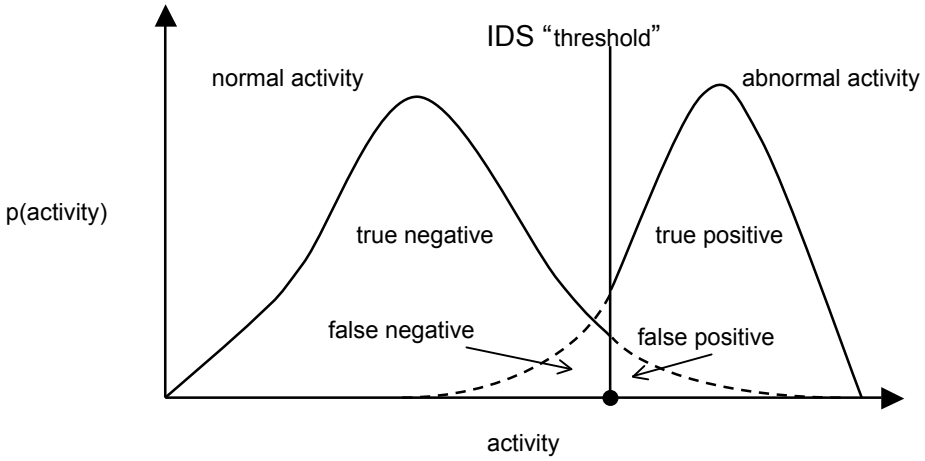


Figure 16 — Compromise between false negatives and false positives

[Dacier 2002] describes various techniques that can be used to reduce the rate of false negatives and false positives, and thus improve the quality of the error reports generated by an IDS. These include using data mining techniques to filter out events with a common root cause, and using diverse error detection mechanisms to increase the proportion of errors detected and the accuracy with which they are detected.

Finally, it is important to note that often while looking for a thing one looks for evidence, side effects, precursors, conduits, and habitats of the thing. As such, the definition of error detection would naturally include vulnerability scanning and configuration checking. In Section 2.5.2, such built-in self-test procedures are termed *pre-emptive error detection*. In the case considered here, an error signal produced by such a background audit procedure would be considered as an input to fault handling and preventive maintenance, rather than as a trigger for automatic error handling.

4.3.2 Error handling

The core dependability concepts of Section 2.5.2 distinguish three forms of error handling:

- **Rollback recovery**: state transformation is carried out by bringing the system back to a previously occupied state, for which a copy (a recovery point, or "checkpoint") has been previously saved.
- **Rollforward recovery**: state transformation is carried out by finding a new state from which the system can operate.
- **Compensation**: state transformation is carried out by exploiting redundancy in the data representing the erroneous state.

In the context of error handling for intrusion-tolerance, examples of each form include:

- **Rollback recovery**:
  - Operating system re-installation
  - TCP/IP connection resets
  - System reboots and process re-initialisation
- **Rollforward recovery**:
  - In threshold-cryptography, replacement of compromised key shares
  - Putting the system into a diminished operation, presumably safe, mode
- **Compensation**:
  - Voting mechanisms
  - Fragmentation-Redundancy-Scattering
  - Sensor correlation

In MAFTIA, the main focus is on compensation techniques for error handling, in particular those listed above, which carry out systematic ***masking*** of intrusions, whereby error compensation is applied even in the absence of intrusions.

4.3.3 Fault handling

In the core dependability concepts, fault handling covers the set of techniques aimed at preventing faults from being re-activated. Whereas error handling is aimed at averting imminent failure, fault handling aims to attack the underlying causes, whether or not error handling was successful, or even attempted. To take a medical analogy: whereas error detection and error handling are concerned with ensuring emergency life support and relieving disease symptoms, fault handling is concerned with curing the disease, or with providing an autopsy.

Section 2.5.2 identifies three fault-handling primitives: fault diagnosis, fault isolation, system reconfiguration and re-initialisation.

4.3.3.1 Fault diagnosis

Fault diagnosis is concerned with identifying the type and locations of faults that need to be isolated before carrying out system reconfiguration or initiating corrective maintenance. This includes faults that are judged to be the cause of detected errors, and faults that could cause problems in the future.

In the case of error signals produced by pre-emptive error-detection mechanisms such as vulnerability-scanners and configuration-checkers, diagnosis is immediate. However, for error

signals from concurrent error-detection mechanisms, it is first necessary to decide whether the underlying cause was an intrusion or an accidental fault.

If the case of intrusions, according to the composite fault model of Section 3.3.2, fault diagnosis can be further decomposed into:

- Intrusion diagnosis, i.e., trying to assess the degree of success of the intruder in terms of system corruption.
- Vulnerability diagnosis, i.e., trying to understand the channels through which the intrusion took place so that corrective maintenance can be carried out.
- Attack diagnosis, i.e., finding out who or what organisation is responsible for the attack in order that appropriate litigation or retaliation may be initiated.

It should be noted that most currently available intrusion-detection systems do not include any fault diagnosis mechanisms. The explicit recognition of the fact that misuses and anomalies are indeed errors that can be caused by any sort of fault is an important result of the MAFTIA project. Indeed, a good intrusion-detection system *requires* such a fault diagnosis mechanism to minimise the rate of false alarms caused by errors due to other classes of faults (e.g., design faults in the reference for defining "misuse" or "anomalies", accidental interaction faults such as mistyping a password, etc.).The data-mining technique described in Chapter 3 of [Dacier 2002] provides a good example of such a fault diagnosis mechanism.

4.3.3.2 Fault isolation

In traditional fault-tolerance, fault isolation is needed, say, to prevent a faulty transmitter from babbling over a shared bus or to prevent a faulty sensor from continuing to add faulty readings to a pool of redundant measurements. That is, we want to make sure that the source of the detected error(s) is prevented from producing further error(s).

In terms of intrusions, this might involve:

- Blocking traffic from an intrusion containment region that is diagnosed as corrupt, by, for example, changing the settings of firewalls or routers
- Removing a corrupted file from the system

or, with reference to the root vulnerability/attack causes:

- Uninstalling software versions with newly-found vulnerabilities
- Arresting the attacker.

4.3.3.3 System reconfiguration and re-initialisation

The occurrence of faults and the consequent isolation of faulty components naturally leads to a decrease in the number of available fault-free resources, so, in traditional fault-tolerant systems, reconfiguration is sometimes envisaged to effectively re-deploy those resources. As already stated in the core dependability concepts, this may mean abandoning some tasks or services (thus resulting in degraded operation) or re-distributing them among the remaining resources.

Reconfiguration of the system allows a possibly degraded service to be delivered while corrective maintenance is carried out on faulty resources. After corrective maintenance, further reconfiguration allows repaired or replacement resources to be re-deployed

In an intrusion-tolerant system possible reconfiguration actions include:

- Software downgrades or upgrades (using appropriate versions are available on-line for this to be done automatically)
- Changing a voting threshold (say from 3-out-of-5 voting to 2-out-of-3 voting) after two corrupt servers have been isolated, so that a further intrusion can be masked

- Deployment of countermeasures including more probes and traps (honey-pots) to gather further information about the intruder, and so assist in attack diagnosis.

4.3.4 Corrective maintenance

Fault handling is usually followed by corrective maintenance, e.g., with the aid of a system security officer or administrator. Such manual intervention is the essential difference between corrective maintenance and (automatic) fault handling. Actions pertaining to corrective maintenance from a security viewpoint include:

- Removing vulnerabilities believed to have contributed to the intrusion:
  - Software revision and upgrade
  - Deployment of security patches
- Attacker rehabilitation.

4.4 Integrated intrusion-detection/tolerance framework

In this section we examine the relationship between intrusion detection, as defined in Sections 4.1 and 4.2, and intrusion tolerance, as examined in Section 4.3. In particular, we will investigate:

- How intrusion detection helps when building an intrusion-tolerant system
- How an intrusion-detection system can itself be made tolerant to faults, including intrusions.

We show how the ideas derived from the core dependability concepts and those from work done by the intrusion-detection community might fit together in a single integrated framework.

Our integrated intrusion-detection/tolerance framework is illustrated on Figure 17.

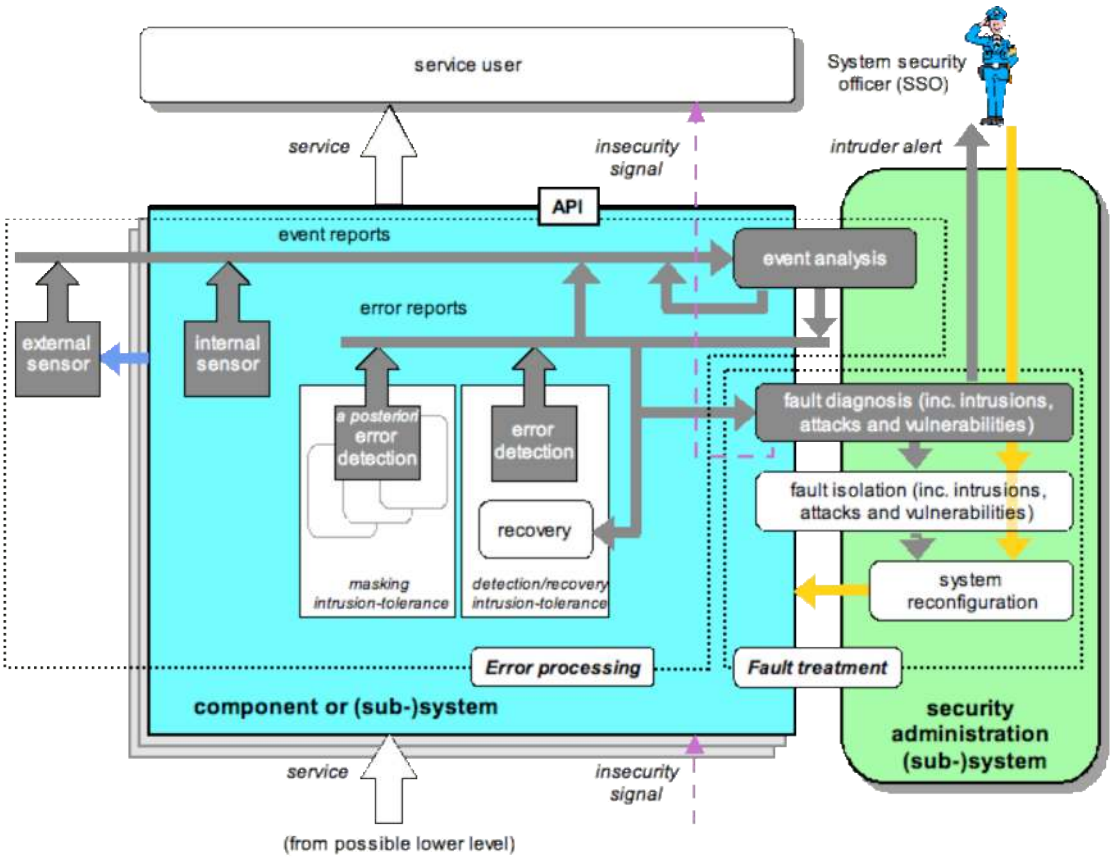


Figure 17 — Integrated intrusion-tolerance framework

The central part of the figure shows a generic MAFTIA component or (sub-)system. There may be many such components within a MAFTIA system, implementing either end-user application functionality or application support services. An administration (sub-)system manages all such components within a single management domain. Here, we consider only the security aspects of system administration within a single management domain. The security-administration component in this diagram spans over all the layers of the system and, in particular, over those comprising the application. The security-administration component is not specific to an individual application but may provide its service to several different applications within the considered management domain.

Components may be layered. The figure shows a component offering some service over an application-programming interface (API) to some higher-level component, using the service(s) offered by possible lower level components. In this case, taking inspiration from the "idealised fault-tolerant component" of [Anderson & Lee 1981], these top and bottom interfaces include "insecurity signals" aimed at informing the service user that the service has been (or might have been) compromised. However, such insecurity signals may not be provided by all generic components, at least not autonomously, since a decision to raise such an insecurity signal may involve some system-wide analysis (by the security administration sub-system).

According to the interpretation of the core fault-tolerance concepts in Section 4.3, we further describe Figure 17 in terms of error detection and error handling, fault handling and corrective maintenance.

4.4.1 Error detection and error handling

We distinguish two basic generic component types:

- Intrusion-intolerant components

- Intrusion-tolerant components

Both component types are potential sources of error-detection information (in the form of event and error reports). However, intrusion-tolerant components are also capable of acting autonomously to implement error recovery.

Error and event reports can be analysed within a given context to confirm or deny suspected errors (cf. Section 4.2.2). Confirmed errors may or may not trigger automatic recovery. In either case, they are reported to the fault handling facilities, which may carry out further analysis toward understanding the root causes of detected errors (fault diagnosis), in order to act thereon (fault isolation and system reconfiguration) and/or report to the system security officer.

4.4.1.1 Intrusion-intolerant components

A central theme of the integrated framework is that any application, service, or layer can be monitored in order to detect deviation from the security policy's description of its correct function (error detection). This monitoring can either be done internally or externally, as portrayed by the "internal sensor" and "external sensor" elements of Figure 17. Monitored components also need to provide context information, which is needed both for error detection (is the suspected error actually an error) and for fault diagnosis (towards accurate classification of faults causing the detected errors).

Internally-monitored components

Placement of error detection and context provision facilities within a component offers several advantages over externally positioned error-detection facilities.

The migration of data between the layers of an application often has a significant computational overhead. By placing error-detection facilities within the components comprising the layers, we eliminate the need to mirror the computation, thereby reducing computational expense, automatically distributing the load, and increasing the accuracy of the view.

While it would be unrealistic to expect all developers to provide specific intrusion-detection features in their code, the use of error-detection facilities is quite common. Many languages provide library facilities to ensure data and process integrity (called assertions). Most code includes some debugging features in the form of logging.

Externally-monitored components

While externally placed error detection and context provision facilities incur greater computational costs and suffer poorer accuracy than their internal counterparts, they are often easier to deploy.

We clarify with an example. Knowledge-based network-based intrusion-detection systems must reconstruct the networking stacks of several different machines looking for signatures (indications) of known attacks. The fact that they must attempt to mirror the process of reconstructing the network stacks of many different machines has several negative implications:

- They have very high computational requirements.
- They must be placed at a location where they are able to observe all traffic that needs to be monitored; this may create networking bottlenecks.
- They have views that may not be identical to the machines they attempt to mirror (packets arriving out of order, dropped packets, etc.).
- Ideally, they should be able to model different implementations of the network stacks (that have different behaviours).

- If a system has to monitor encrypted traffic, it must have a set of virtual master keys (this is potentially dangerous as it is a single point of confidentiality failure for the network) and have the capacity to perform the necessary decryption (which is certainly expensive).

On the other hand, network-based intrusion-detection systems are comparatively easy to deploy and maintain.

4.4.1.2 Intrusion-tolerant components

The second important type of generic MAFTIA components consists of those that provide internal recovery to errors caused by intrusions. Such components may implement fault-tolerance using either error detection and (rollback or rollforward) recovery, or intrusion-masking (cf. Section 4.3.2). In MAFTIA, particular attention had been given to the latter variety of intrusion-tolerance, e.g., using the FRS technique, which can compensate errors due to both accidental faults and intrusions [Fraga & Powell 1985]. Possible applications of this approach include services based on *trustworthy* trusted third parties such as those described in [Abghour et al. 2001, Cachin 2001a]:

- Certification authority and directory service
- Fair exchange TTPs
- Notary service
- Authentication service
- Authorisation service

or indeed, sub-components of an intrusion-detection service (e.g., intrusion-tolerant sensor correlation and event analysis).

Whether masking or detection-and-recovery is used, detected errors and other relevant events are analysed and reported to the fault handling facilities. Intrusion-tolerant components are thus a particular kind of internally-monitored components.

4.4.2 Fault handling

The fault-handling facilities include the means for diagnosing and isolating faults (including intrusions, attacks and vulnerabilities), and for automatic or manual system reconfiguration (cf. Section 4.3.3). Whereas it seems feasible to internally implement some degree of fault diagnosis and isolation within the considered component (this would be necessary if the component were to be capable of autonomously raising an insecurity signal), it is expected that it will often be necessary to take into account a more system-wide view. Moreover, such a system-wide view seems essential to carry out meaningful system reconfiguration. For these reasons, Figure 17 shows the fault diagnosis and isolation mechanisms distributed across the generic component(s) and the security administration system, whereas the system reconfiguration mechanisms are internal to the latter, which may possibly be distributed.

From the viewpoint of intrusion-detection, the IDS (as defined in Section 4.1, i.e., excluding the so-called response mechanisms) within this integrated framework consists of the set of external and internal sensors, the error-detection mechanisms of any intrusion-tolerant components, and the event analysis and fault diagnosis mechanisms that signal intruder reports to a system security officer. These are shown in dark grey on Figure 17.

4.4.3 Corrective maintenance

In many cases, manual intervention by the SSO will be needed to complete automatic fault-handling. Since the number of false positives in practical intrusion detection systems is non-negligible, the SSO will usually have to be involved to judge between true and false positives.

If he decides that an alarm is a true positive, he may trigger further countermeasures over and above any that have been triggered automatically. If the number of false positives is judged to be too high, then the SSO can carry out corrective maintenance on the intrusion detection system by, for example, adding appropriate filtering rules. Similarly, if the SSO realises that the IDS is failing to detect attacks and intrusions for some reason, i.e., the rate of false negatives is too high, then additional sensors and correlation rules can be added. See Chapters 3 and 4 of [Dacier 2002] for more details of the techniques developed by MAFTIA for improving the quality of intrusion detection by reducing the number of false positives and false negatives.

4.4.4 Relationship between error detection, fault handling, and corrective maintenance

The following diagram, Figure 18, is intended to clarify the relationship between error detection by the IDS, fault handling by the SSO, and corrective maintenance of the IDS. The target system shown in Figure 18 could represent a single component or sub-system, or an entire computer network that is being monitored by an IDS. The error reports generated by the sensors and event analysers of the IDS are passed to the security administration system and the SSO for fault diagnosis. Based on the results of this diagnosis, the SSO determines whether the appropriate response is to do nothing (e.g., because the observed attack is not successful), or to reconfigure the IDS to improve the quality of the error detection (e.g., because the rate of false positives is too high), or to reconfigure the target system, (e.g., to remove a vulnerability that has been detected). In this respect, the combination of the SSO and the security administration system partially fulfil the role of the r-box in the CIDF model (although as noted earlier in Section 4.2, we do not consider the r-box to be part of intrusion detection *per se*, but rather part of the set of facilities providing error recovery, fault isolation and system reconfiguration in a general intrusion-tolerance framework).

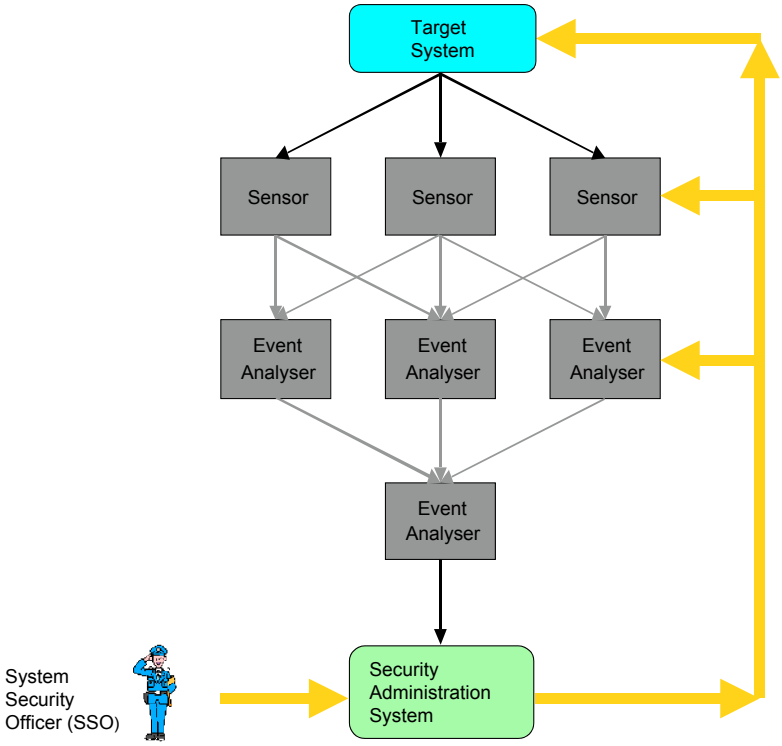


Figure 18 — Role of SSO in error detection, fault diagnosis and corrective maintenance

4.4.5 An illustrative example

As an example of how intrusion-detection and intrusion-tolerance fit together, Figure 19 shows a much simplified, unfolded interpretation of Figure 17.

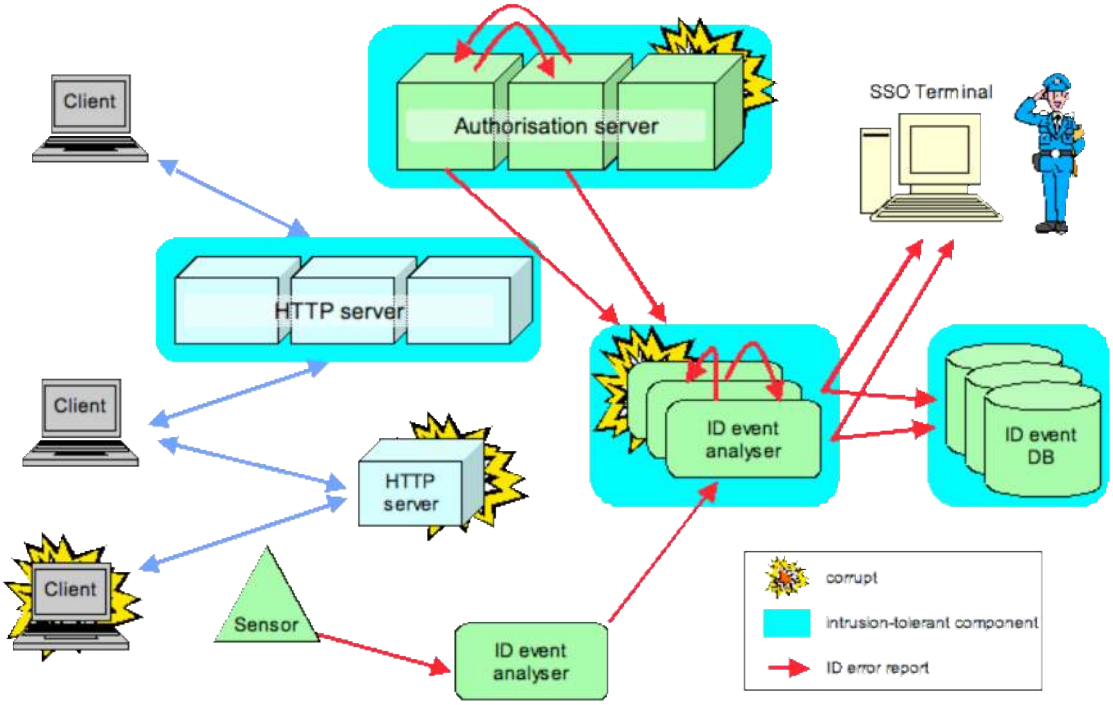


Figure 19 — Relationship between intrusion-detection and intrusion-tolerance

Figure 19 shows some typical elements of a MAFTIA system, including both intrusion-intolerant and intrusion-tolerant components, for both application and system services (authorisation and intrusion-detection are viewed here as system services).

In this example, the system contains a fault-tolerant web server and fault-tolerant authorisation server, both capable of masking intrusions and signalling any detected errors. There is also a fault-intolerant web server, monitored by an external sensor.

All error-detection sources can produce intrusion-detection (ID) error reports. For clarity, only those due to corrupt components are shown. The ID error reports are sent through a chain of two ID event analysers (cf. Figure 13, page 44) to an ID event database and to the system security officer. Certain components specific to the IDS (one of the two event analysers and the ID event database) are also fault-tolerant, and are thus capable of being themselves sources of ID error-reports.

For simplicity, the example does not distinguish event analysers aimed at confirming suspected errors from those aimed at diagnosing faults. Nor does the figure portray any automatic reconfiguration logic, i.e., it is assumed in this example that any reconfiguration would be carried out under manual control of the SSO.

Chapter 5 Architectural overview

The purpose of this chapter is to introduce the basic models and assumptions underlying the design of the MAFTIA architecture, and then to present an overview of the architecture itself from various perspectives. The discussion of models and assumptions is of course informed by the previous material in Chapters 3 and 4, which explains security notions of intrusion, attack, and vulnerability in terms of the more classical dependability concepts of faults, failures and errors, and then outlines the basic MAFTIA approach towards building an intrusion-tolerant architecture through the use of intrusion-detection systems and intrusion-tolerant components. This chapter details both the functional aspects of the architecture, and the constructs aimed at achieving intrusion tolerance. It concludes with some examples of how the architecture will be used to build intrusion-tolerant services.

5.1 On the nature of trust

The adjectives "trusted" and "trustworthy" are central to many arguments about the dependability of a system. In the security literature, the terms are often used inconsistently. For example, Anderson [Anderson 2001] points to differing usages of the notions of "trust":

- U.S. National Security Agency (NSA) definition: "A trusted system or component is one whose failure can break the security policy, while a trustworthy system or component is one that won't fail".
- U.K. military view: a trusted system element is one "whose integrity cannot be assured by external observation of its behaviour while in operation".
- Other definitions which have to do with whether a particular system is approved by an authority: "A trusted system won't get me fired if it's hacked on my watch", or even "a system we can insure".

The MAFTIA notions of "trust" and "trustworthiness" are a generalization of the NSA notions: they point to generic properties and not just security; and there is a well-defined relationship between them — in that sense, they relate strongly to the words "dependence" and "dependability".

As already noted in Annex B to Chapter 2, the term *trustworthiness* is essentially synonymous to dependability, but is often the preferred term when the focus is on external faults such as attacks.

Trust is the reliance put by a component, on some properties (functional and/or non-functional) of another component, subsystem or system[13].

In consequence, a *trusted component* has a set of properties that are relied upon by another component (or components), i.e., there is an *accepted dependence*. If A trusts B, then A accepts that a violation in those properties of B might compromise the correct operation of A. Note that trust is not absolute: the *degree of trust* placed by A on B is expressed by the set of properties, functional and non-functional, which A trusts in B (for example, that a smart card gives a correct signature for every input, with a certain MTTF for a given level of threat).

Observe that those properties of B trusted by A might not correspond quantitatively or qualitatively to B's actual properties. Thus, A should only trust B *to the extent of* B's trustworthiness. In other words, trust, the belief that B is dependable, should be placed in the measure of B's dependability.

[13] We will just use 'component' henceforth, for simplicity. Likewise, this relation can be generalised to collections of components.

The trustworthiness of a component is, not surprisingly, a measure of the extent to which a component, subsystem or system, meets a set of properties (functional and/or non-functional). The trustworthiness of a component can be derived from its construction, and/or evaluated as appropriate. For example, a smart card used to implement the example above should actually meet or exceed the required properties.

The definitions above have obvious (and desirable) consequences for the design of intrusion tolerant systems: trust is not absolute, it may have several degrees, quantitatively or qualitatively speaking; it is related not only with security-related properties but with any properties (e.g., timeliness); trust and trustworthiness lead to complementary aspects of the design and verification process. In other words, when A trusts B, A *assumes* something about B. The trustworthiness of B measures the *coverage* of that assumption.

In fact, one can reason separately about trust and trustworthiness. One can define chains or layers of trust, make formal statements about them, and validate this process. To complement this line of reasoning, one should also ensure that the components involved in the above-mentioned process are endowed with the necessary trustworthiness. This alternative process is concerned with the design and verification of new components, or the verification/certification of existing ones (e.g., COTS). The two terms, trust and trustworthiness, establish a separation of concerns on the failure modes: of the higher level algorithms or assertions (e.g., authentication/authorization logics); and of the infrastructure running them (e.g., processes/servers/communications).

Let us analyze how to build justified trust under this model. Assume that the trustworthiness of component C is defined in terms of some predicate P that holds with a coverage Pr. Another component B should thus trust C to the exact extent of C possessing P with a probability Pr, not more, not less. So, although there can be failures consistent with the limited trustworthiness of C (i.e., that $Pr < 1$), these are "normal", and who/whatever depends on C, like B, should be aware of that fact, and expect it.

However, it can happen that B trusts C to a greater extent than it should: trust was placed on C to an extent greater than its trustworthiness, perhaps due to an incorrect or negligent perception of the latter. This is a mistake of who/whatever uses C, which can lead to unexpected failures.

Finally, it can happen that the claim made about the trustworthiness of C is wrong (about predicate P, or its coverage Pr, or both). The component fails in worse, earlier, or more frequent modes than stated in the claim made about its resilience. In this case, even if B trusts C to the extent of satisfying predicate P with probability Pr, there can still be unexpected failures. However, this time, the failures will be due to a mistake of whoever architected/built the component C.

The intrusion-tolerance strategies adopted within MAFTIA rely upon these notions. The assertion 'trust on a trusted component' inspires the following guidelines for the construction of modular fault tolerance in complex systems: components are trusted to the extent of their trustworthiness; there is separation of concerns between what to do with the trust placed on a component (e.g., building fault-tolerant algorithms), and how to achieve or show its trustworthiness (e.g., constructing the component). The practical use of these guidelines is exemplified in later sections.

5.2 Models and assumptions

5.2.1 Failure assumptions

A crucial aspect of any fault-tolerant architecture is the fault model upon which the system architecture is conceived, and component interactions are defined. The fault model conditions the correctness analysis, both in the value and time domains, and dictates crucial aspects of system configuration, such as the placement and choice of components, level of redundancy,

types of algorithms, and so forth. A system fault model is built on assumptions about the way system components fail. Classically, these assumptions fall into essentially two kinds: *controlled failure* assumptions, and *arbitrary failure* assumptions.

Controlled failure assumptions specify qualitative and quantitative bounds on component failures. For example, the failure assumptions may specify that components only have timing failures, and that no more than f components fail during an interval of reference. Alternatively, they can admit value failures, but not allow components to spontaneously generate or forge messages, nor impersonate, collude with, or send conflicting information to other components. This approach is realistic, since it represents very well how common systems work under the presence of accidental faults, failing in a benign manner most of the time. It can be extrapolated to malicious faults, by assuming that they are qualitatively and quantitatively limited. However, it is traditionally difficult to model the behaviour of a hacker, so we have a problem of coverage that does not recommend this approach unless a solution can be found.

Arbitrary failure assumptions ideally specify no qualitative or quantitative bounds on component failures. Obviously, this should be understood in the context of a universe of "possible" failures of the concerned operation mode of the component. For example, the possible failure modes of interactions between components of a distributed system might be limited to combinations of timeliness, form, meaning, and target of those interactions (let us call them messages), and might not encompass the arbitrary cloning of system components. In this context, an arbitrary failure means the capability of generating a message at any time, with whatever syntax and semantics (form and meaning), and sending it to anywhere in the system. Practical systems based on arbitrary failure assumptions must however specify quantitative bounds on the number of failed components, or at least equate tradeoffs between resilience of their solutions and the number of failures eventually produced [Babaõglu 1987]. Arbitrary failure assumptions are costly to handle, in terms of performance and complexity, and thus are not compatible with the user requirements of the vast majority of today's on-line applications.

Hybrid assumptions combining both kinds of failure assumptions would be desirable. They provide a known framework in dependable system design vis-à-vis accidental failures. Generally, they consist of allocating different assumptions to different subsets or components of the system, and have been used in a number of systems and protocols [Meyer & Pradhan 1987, Powell et al. 1988].

5.2.2 Composite fault model

With hybrid assumptions some parts of the system would be justifiably assumed to exhibit fail-controlled behaviour, whilst the remainder of the system would still be allowed an arbitrary behaviour. This would be advantageous in modular and distributed system architectures such as MAFTIA.

However, such an approach is only feasible when the fault model is well-founded, that is, the behaviour assumed for every single subset of the system can be modelled and/or enforced with high coverage. As a matter of fact, a system normally fails by its weakest link, and naïve assumptions about a component's behaviour will be easy prey to hackers.

As we have discussed in Chapter 3, the impairments that may occur to a system, security-wise, have to do with a wealth of causes, which range from internal faults (e.g. vulnerabilities), to external, interaction faults (e.g., attacks), whose combination produces faults that can directly lead to component failure (e.g., intrusion).

A first step towards our objective is the organisation of these diverse causes into a composite fault model (cf. Figure 8, page 31), with a well-defined relationship between attack/vulnerability/intrusion. Such a model allows us to modularise our approach to

achieving dependability, by combining different techniques and methods tackling the different classes of faults defined. Ten such security methods were defined in Section 3.4.

5.2.3 Enforcing hybrid failure assumptions

The second step is the enforcement of hybrid failure assumptions. A composite fault model with hybrid failure assumptions is one where the presence and severity of vulnerabilities, attacks and intrusions varies from component to component. There is a body of research, starting with [Meyer & Pradhan 1987] on hybrid failure models that assume different failure type distributions for different nodes. For instance, some nodes are assumed to behave arbitrarily while others are assumed to fail only by crashing. The probabilistic foundation of such distributions might be hard to sustain in the presence of malicious intelligence, unless their behaviour is constrained in some manner. Our work might best be described as *architectural hybridization*, in the line of works such as [Powell et al. 1988] and [Verissimo et al. 1997], where failure assumptions are in fact enforced by the architecture and the construction of the system components, and thus substantiated.

Consider a component or sub-system for which a given controlled failure assumption was made. How can we achieve coverage of such an assumption, given the unpredictability of attacks and the elusiveness of vulnerabilities?

The answer lies in the approach taken to the design, construction and/or configuration of the component. Through the combined use of intrusion prevention techniques (i.e. attack and vulnerability prevention and removal), and ultimately the implementation of internal intrusion-tolerance mechanisms, we must justifiably achieve confidence that the component behaves as assumed, failing in a controlled manner, i.e., that the component can be *trusted* (cf. Section 5.1) because it is *trustworthy*. The combination of these techniques should be guided by the composite fault model mentioned above (i.e., removing vulnerabilities that are matched by attacks we cannot prevent; preventing or tolerating attacks on vulnerabilities we cannot remove, etc.). The measure of this trust is the coverage of the controlled failure assumption.

Looking at the next higher level of abstraction — the level of the system — we are now ready to implement our intrusion-tolerance mechanisms, using a mixture of arbitrary-failure (fail-uncontrolled or non trusted) and fail-controlled (or trusted) components. However, our task is made easier since the controlled failure modes of some components vis-à-vis malicious faults restrict the system faults the component can produce. In fact we have performed a form of *fault prevention* at the system level: some kinds of system faults are simply not produced.

5.2.4 Intrusion tolerance under hybrid failure assumptions

The approach outlined in the previous sections:

- establishes a divide-and-conquer strategy for building modular fault-tolerant systems, with regard to failure assumptions;
- can be applied to achieve different behaviours in different components;
- can be applied recursively at as many levels of abstraction as are found to be useful.

Consider the discussion of Section 5.1 on trust and trustworthiness as being synonyms of dependence and dependability. Taking the dependability argument further, in MAFTIA we trust components or subsystems (we will just use the word component henceforth) *to the extent of* their trustworthiness, as perceived at the adequate instances: by the designer, tester, reviewer, user (human or another component), etc.

That is, trustworthy components are components whose coverage has been justified, either by argumentation concerning the techniques used in their implementation or through

quantification by some attack and vulnerability forecasting methods[14] (cf. Section 3.4), can subsequently be used in the construction of fault-tolerant protocols under architectural hybrid failure assumptions. In properly designed systems, the trust placed on a component should be qualitatively and/or quantitatively commensurate to its trustworthiness. This is an innovative aspect in MAFTIA that we explore in the following sections, and the fact that it is not yet known very well how to provide quantitative arguments in the context of malicious fault metrics, presents an opportunity, rather than a threat to this approach.

Note that the soundness of the approach does not depend on our making possibly naïve assumptions about what a hacker can or cannot do to a component. Instead, we analyse and break the attack/vulnerability/intrusion chain selectively, removing vulnerabilities that match attacks we cannot prevent, preventing attacks that exploit vulnerabilities we cannot remove, and/or finally tolerating any intrusions on the component that we cannot prevent with the above methods.

This approach is explored in several ways within MAFTIA. In particular, it is our rationale for implementing small trustworthy components (in the sense discussed in Section 5.1), which are then trusted by other components. Such components are simple enough to be built and plausibly shown to be correct. This allows us to construct implementations of fault-tolerant protocols that are more efficient than protocol implementations that have to deal with truly arbitrary assumptions, and more robust than designs that make controlled failure assumptions without enforcing them.

There are three main instances of such trusted components that we describe in more detail in the subsequent architecture overview (see Section 5.3.4). The first is based on a Java Card, and is a local component designed to assist the crucial steps of the execution of services and applications. The second is a distributed component (named Trusted Timely Computing Base), based on appliance boards with private network adapters, which is designed to assist crucial steps of the operation of middleware protocols.

We use the word “crucial” in both instances to stress the tolerance aspect: unlike prevention-based approaches (e.g., classical Reference Monitors), the trusted component does not mediate all accesses to resources and operations. In our approach, protocols run in an untrusted environment, and local participants only trust interactions with the trusted components (and only to the extent of their trustworthiness, as will become clear later on).

The local trusted component is used to certify certain operations, through public key cryptography. It is a valuable assistant, for example, of protocols supporting the high-level services of the middleware, such as the authorisation or transactional support services.

The distributed trusted component is used to assist group communications and group activity protocols. It provides simple security functions (mainly secure IPC channels between itself and any local component) and a distributed consensus function on simple facts of protocol operation. It also provides time-related functions that will be discussed in the next sections.

Whereas these two instances could be best seen as low-level runtime support components, the third instance concerns the recursive view of building macroscopic, distributed trusted components in the middleware. Given a hostile environment, single components, including networks, can be corrupted, and higher level components engaging in distributed activities might benefit from trusting middleware components to provide a set of correct support services, whose provision is built on distributed fault-tolerance mechanisms, for example through agreement and replication amongst collections of participants in several hosts.

[14] Such quantification is currently beyond the state-of-the-art, and is not being addressed in MAFTIA.

5.2.5 Arbitrary failure assumptions considered necessary

Notice that the hybrid failure approach, no matter how resilient, relies on the coverage of the fail-controlled assumptions. Definitely, there will be a significant number of operations in the kind of applications to be served by MAFTIA, whose value and/or criticality is such that the risk of failure due to violation of these assumptions cannot be incurred.

In consequence, an important area of research being pursued is related with arbitrary-failure resilient building blocks, namely communication protocols of the Byzantine class, which do not make assumptions on the existence of trusted components or other fail-controlled components. They reason in terms of admitting any behaviour from the participants, and allow the corruption of a parameterisable number of participants, say *f*. The system works correctly as long as there exist $n>3f$ participants.

These protocols do not make assumptions about timeliness either, and are in essence time-free. This has implications on the operational aspects, which will be further discussed in the next sections.

5.2.6 Synchrony models

Research in distributed systems algorithms has traditionally been based on one of two canonical models: fully asynchronous and fully synchronous models [Verissimo et al. 2000]. In this section, we discuss the limitations of both models, in order to motivate the hybrid approach that MAFTIA is taking.

Asynchronous models are time-free, that is, they are characterised by an absolute independence of time, and distributed systems based on such models typically have the following characteristics:

Pa 1 Unbounded or unknown processing delays

Pa 2 Unbounded or unknown message delivery delays

Pa 3 Unbounded or unknown rate of drift of local clocks

Pa 4 Unbounded or unknown difference of local clocks[15]

Asynchronous models obviously resist timing attacks, i.e., attacks on the timing assumptions of the model, which are non-existent in this case. Because of this fact, they enjoy a resilience that is not shared by synchronous models, and which is a crucial asset in the presence of malicious faults. However, for some time, asynchronous models were not much considered in the literature due to a belief that there could only be inefficient solutions to many interesting problems, such as consensus or Byzantine agreement. In addition, fully asynchronous models preclude the deterministic solution of those problems. “False” asynchronous algorithms have been deployed over the years, exhibiting subtle but real failures, thanks to the inappropriate use of timeouts in a supposedly time-free model.

Work in MAFTIA takes new approaches to this problem, showing innovative efficient solutions through probabilistic asynchronous protocols [Cachin 2001b]. It does not matter that such solutions are only probabilistic as long as the error probability can be made sufficiently small for the applications in view (in particular smaller than the probability of hardware faults, etc.).

However, because of their time-free nature, asynchronous models cannot solve timed problems. In practice, many of the emerging applications we see today, particularly on the Internet, have interactivity or mission-criticality requirements. Timeliness is part of the

[15] **Pa3** and **Pa4** are essentially equivalent but are listed for a better comparison with the synchronous model characteristics listed below. Since a local clock in a time-free system is nothing more than a sequence counter, clock synchronisation is also impossible in an asynchronous system.

required attributes, either because of user-dictated quality-of-service requirements (e.g., network transaction servers, multimedia rendering, synchronised groupware, stock exchange transaction servers), or because of safety constraints (e.g., air traffic control). In contrast to asynchronous models (which simply have no notion of time) synchronous models allow timeliness specifications. In this type of model, it is possible to solve all the typical hard problems deterministically (e.g., consensus, atomic broadcast, clock synchronisation) [Chandra & Toueg 1996]. Synchronous models have the following characteristics:

Ps 1 There exists a known bound for processing delays by non-faulty processors

Ps 2 There exists a known bound for message delivery delays between non-faulty processors

Ps 3 There exists a known bound for the rate of drift of non-faulty local clocks

Ps 4 There exists a known bound for the difference among non-faulty local clocks

In consequence, such models solve timed problem specifications, one precondition for at least a subset of the applications targeted in MAFTIA, for the reasons explained above. Imagine for example the technical difficulty of implementing real-time stock exchange transactions on the Internet, based on real-time quotes, and with temporal order between competitive requests, to ensure market fairness.

However, synchronous models are fragile in terms of their coverage of timeliness assumptions such as positioning of events in the timeline or determining execution durations. It is easy to see that synchronous models are susceptible to timing attacks, since they make strong assumptions about things happening on time. For example, algorithms based on messages arriving by a certain time, or on reading the actual global time from a clock, or on securing the temporal order of messages, may fail in dangerous ways if manipulated by an adversary [Gong 1992]. In a synchronous setting, the difficulty of implementing real-time stock exchange transactions over the Internet in the presence of malicious faults could become insurmountable.

Work in MAFTIA takes the *timed partially synchronous* approach to this problem. The intermediate synchrony model we follow provides a solution to the problems enumerated above, essentially for three reasons: (i) it allows timeliness specifications; (ii) it admits failure of those specifications; (iii) it provides timing failure detection, and if desired, timing fault tolerance.

To summarise, a time-free approach is necessary when the criticality of operations is such that an arbitrary failure assumptions model is needed to maximize coverage and prevent timing attacks by resorting to an asynchronous model. However, this setting does not offer timeliness guarantees and that would be the price to pay. The hybrid approach that we are taking in MAFTIA, which we now discuss in more detail, attempts to improve on this situation.

5.2.7 Timed approach

Let us analyse a little more how timed algorithms can be attacked. Specifying timeout values may be very difficult when protecting against arbitrary failures that may be caused by a malicious attacker. It is usually much easier for an intruder to attack communication with a server than to subvert the server itself. Even asynchronous systems with failure detectors [Chandra & Toueg 1996] can easily be fooled into having inconsistent and wrong failure suspicions about honest parties. This problem arises because the failure detector is built on the assumption that the system will be stable for long enough periods. This assumption may obviously fail against a malicious adversary. Two possible solutions present themselves: either the failure detector is made to work properly in a malicious fault environment, or a solution is devised that does not require failure detectors. We will address the latter in the next section.

As for the former, we adopt a partially synchronous model, enriched with the notion of a *timing failure detector*. This is a stronger definition of detector than the crash failure detector. However, the power of such a detector addresses our concerns about timeliness. We expect our timed applications to be able to run in environments of uncertain synchrony, such as the Internet. Thus, in spite of having a notion of timeliness (i.e., time bounds, deadlines, etc.), they may not always be able to fulfil these requirements adequately. Consequently, we assume that components can exhibit timing failures, i.e., they can violate timeliness properties. This would only be dangerous if we were not able to detect them, otherwise we can devise timing-fault tolerant protocols. Thus, we require our timing failure detector to be resilient to malicious faults: it will not make mistakes even in the presence of intruders. This addresses the concerns expressed at the beginning of this section.

The realisation of our model is called the *Trusted Timely Computing Base (TTCB)*: an architectural device working as an oracle performing timing failure detection, built in a way so as to ensure detection is timely, accurate and complete [Verissimo et al. 2000], even in the presence of malicious faults. The TTCB must be: distributed, for detection to work correctly system-wide; synchronous, so that timing operations are accurate; and fail-controlled, to provide well-defined behaviour in the presence of intrusions. In the context of the discussion of Section 5.2.4, the TTCB is built as a distributed and synchronous trusted component, which provides useful security-related functions alongside time-related functions, and is used to support the construction and operation of fault-tolerant protocols following the timed approach.

In a sense, a TTCB might sound similar to the very well known paradigm in security of a Trusted Computing Base (TCB) [Abrams et al. 1995]. However, the objectives are radically different. A TCB aims at fault prevention and ensures that the whole application state and resources are tamper-proof. Furthermore, it is based on logical correctness and makes no attempt to reason in terms of time. In contrast, a TTCB aims at *fault tolerance*: it simplifies the task of application components, but most of the application code and state is in unprotected space, and can be tampered with, requiring the use of redundancy so that the whole application does not fail. In other words, a TTCB significantly reduces the part of the system about which tamper-proofness claims need be made. It is an architectural artefact supporting the construction and trusted execution of intrusion-tolerant protocols and applications running under a partially synchronous model.

This type of hybrid fault model allowed us to devise a new Byzantine-resilient reliable multicast protocol for asynchronous systems [Abrams et al. 1995]: we assume that the TTCB can only fail by crashing, while the rest of the system can behave in a Byzantine way. By relying on the services of the TTCB, the protocol exhibits excellent behaviour in terms of time and message complexity when compared with more traditional Byzantine protocols. Moreover, it only requires $n \geq f + 2$ correct processes to tolerate f failures, instead of the usual $n \geq 3f + 1$.

5.2.8 Time-free approach

The time-free approach taken in MAFTIA adopts the asynchronous model. Of course, asynchronous protocols cannot guarantee a bound on the overall response time of an application, but they were never meant to. In general, an asynchronous model provides a conceptually simple and nice framework for developing and reasoning about the correctness of an algorithm, satisfying safety under any conditions, and providing liveness under certain conditions, which in MAFTIA asynchronous protocols are defined in a probabilistic way. This has some advantages for the design of secure distributed systems, which is one reason for pursuing such a model in the context of MAFTIA. In fact, sometimes it is necessary and worthwhile to sacrifice timeliness for resilience, for example for very critical operations (key distribution, contract signing, etc.)

In the asynchronous model, consensus is not reachable by deterministic protocols, even with crash failures only. But there are randomised solutions that use only a constant number of rounds to reach agreement [Bracha & Toueg 1985, Rabin 1989]. In MAFTIA, by employing modern, efficient cryptographic techniques, this approach has been extended to a practical yet provably secure protocol for Byzantine agreement in the cryptographic model that withstands the maximal possible corruption [MAFTIA 2000]. The cryptographic model with randomised Byzantine agreement is both practically and theoretically attractive. Randomised agreement protocols may not terminate with non-zero probability, but this probability can be made negligible. In fact, a protocol using cryptography always has a residual probability of failure, determined by the key lengths. In consequence, this is a solution that works under arbitrary failure assumptions, that is, faults (attack, intrusions) both in the time and space domains.

We have observed that randomised (probabilistic) protocols like Byzantine agreement make essentially very few assumptions about the environment. One possible track in the quest for more efficient implementations close to the boundary of arbitrary failure assumptions would be to assume two operation modes. The optimistic asynchrony model that we are pursuing in the MAFTIA project attempts to address this track. A fully asynchronous model is assumed as a baseline framework, running randomised Byzantine agreement. However, whenever the system exhibits enough synchrony, the system switches to a partially synchronous operation mode, still malicious-fault resilient, but exhibiting better performance. The TTCB could be used to make the algorithms and protocols aware of the current synchrony of the system, thus enabling them to change operation mode in an accurate way.

5.2.9 Programming model

Although the main goal of MAFTIA is to provide security in the face of malicious faults, the architecture must also provide a versatile functional support in order to be useful. Consequently, it will support the main interaction styles used in distributed computing, namely:

- *client-server*, for service invocations
- *multipeer*, for interactions amongst peers
- *dissemination*, of information in push or pull form
- *transactions*, for encapsulation of multiple actions

Client-server interactions can be implemented by two different mechanisms: in *closed loop*, usually performed through RPC, or in *open loop*, usually performed through group communication. Both approaches are easily implemented using group-based open-loop mechanisms, such as offered by the middleware. Another style of interaction is *multipeer*, conveying the notion of spontaneous, symmetric interchange of information, amongst a collection of peer entities. Multipeer interactions are the kind of interaction one might wish among managers of a distributed database, a group of commerce servers, a group of TTP servers, or a group of participants running a cryptographic agreement protocol (e.g., contract signing). Next, we have *dissemination*, which combines the information push and pull approaches. Information is published by *publishers*, and is made available to interested *subscribers*. Message subscription can be implemented using two different alternatives: the *push* strategy or the *pull* strategy. Finally, *transactions* provide the capability of performing sets of operations atomically, i.e. satisfying the well-known ACID properties.

The various styles referred to above can be combined to form more complex interaction styles. For example, transactions may encapsulate several interactions built using the other styles. Note also that the extensive use of open-loop client server mechanisms, multipeer interactions, replication, and distributed transactions is yet another justification for the emphasis of the group-orientation paradigm in the architecture of MAFTIA.

5.3 Architecture

In this section, we provide an overview of the MAFTIA architecture and discuss the various options that it offers at the hardware, local executive and distributed software levels.

5.3.1 Overview

The MAFTIA architecture is highly modular. This is an accepted design principle for building distributed fault tolerance into systems. It facilitates the definition of different redundancy strategies for different components, and the placement of the relevant replicas.

MAFTIA also aims at applications with a geographically large scale, namely services provided to many clients coming from very far apart, whose core part may run on several, possibly interconnected facilities of one or more organisations. Most of the research work in MAFTIA is devoted to the design of suitable middleware protocols to ease the construction of the core part of such services, and to the development of the services themselves and of their interaction with clients. With regard to scalability, these protocols will, whenever appropriate, be *topology aware*, a powerful construct for designing large-scale efficient protocols [Rodrigues & Verissimo 2000].

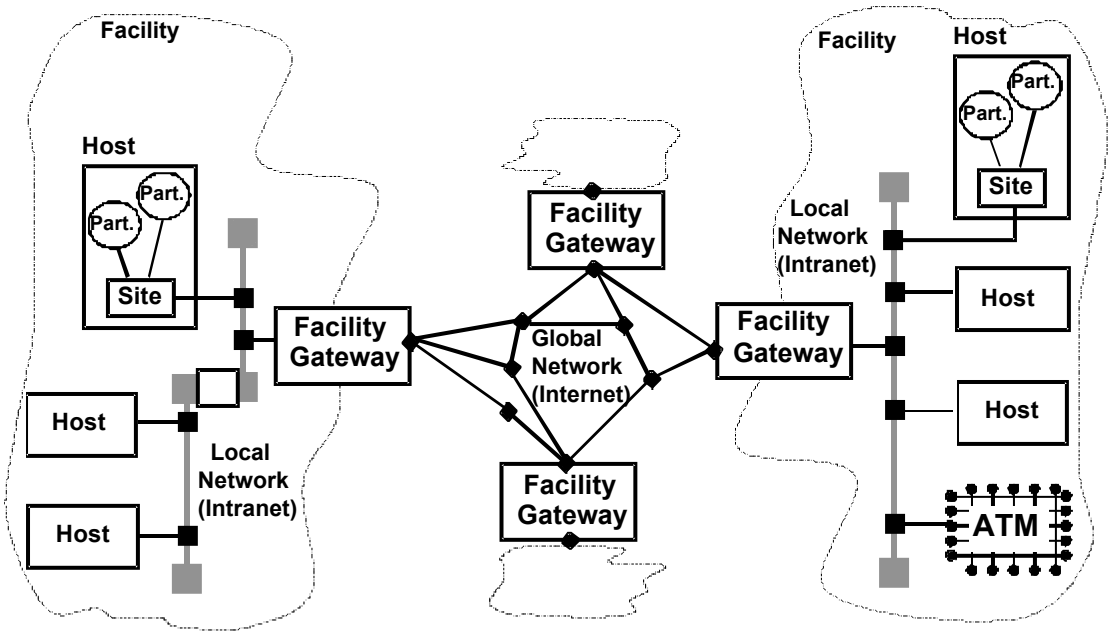


Figure 20 — Two-tier WAN-of-LANs

For example, at global level, there is advantage in recognising the topology of the networking infrastructure as a logical two-tier *WAN-of-LANs*, as suggested in Figure 20: *facilities* composed of pools of hosts (intranets) with privately managed high connectivity links, such as LANs or MANs or ATM fabrics, are normally interconnected in the upper tier by the publicly managed point-to-point global network (the Internet), through *facility gateways,* logical devices that represent the local network members for the global network. Such gateways not only serve as clustering points in terms of scale, but may also serve as intrusion prevention devices, creating error containment domains (fire walling; inspecting incoming and outgoing traffic for attack and intrusion detection; ingress and egress traffic filtering; internal topology hiding, etc.).

As a matter of fact, such a structure offers opportunities for making different assumptions regarding the types and levels of threat and degrees of vulnerability of the local network versus the global network part. This does not necessarily mean considering intra-facility networking threat-free. For example, certain port scans or pings in the global network may be

completely innocent and harmless, whereas they may mean an attack if performed inside the facility. Likewise, an intruder working from the inside of the facility may have considerably more power than one working from the outside. In a global information society as considered in MAFTIA, many participants will be coming from individual access points and not from organisations: tax payers, voters, money owners, e-commerce customers, etc. They will mostly be clients of MAFTIA services. Clearly, the WAN-of-LAN structure is expected to be helpful in organising the latter, whereas clients will normally interact with the services through simple and mostly standard interfaces.

The WAN-of-LANs view we have just presented can be recursively applied, in order to represent very-large-scale organisations. On an intra-facility level, further hierarchies, namely those already deriving from hierarchical organisation of sub-networks and domains, are not precluded. On an intra-organisation (multi-facility) level, the topology depicted in Figure 20 can be re-instantiated to represent an organisation with multiple geographically dispersed facilities interconnected by secure tunnels whose end points are internal Facility Gateways, whose sole role is to implement the Virtual Private Network (VPN) interconnecting all organisation facilities.

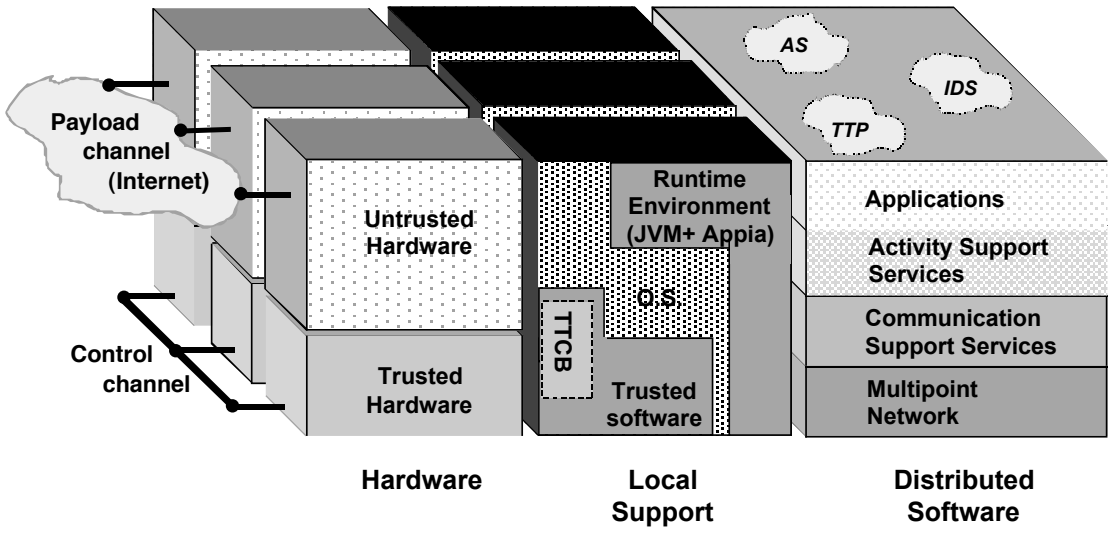
On the other hand, inside a host, we make a separation between the functionality concerned with inter-host communication, which we call *site* level functions, and the functionality concerned with distributed activity of processes, tasks, objects, etc., which we call *participant* level functions (see Figure 20). Participants, which execute distributed activities, can be senders or recipients of information, or both, in the course of the aforementioned activities. For example, if more than one participant residing on a host is a recipient of a reliable multicast message, the relevant group communication protocol runs at site level and only delivers one message at that host, which is copied to all local recipients. From now on, when specifying operations inside or among hosts in MAFTIA, we will refer to sites when taking the communication/networking viewpoint on the system, and we will refer to participants, when taking the activity/processing viewpoint.

5.3.2 Main architectural options

The structure of a MAFTIA host relies on a few main architectural options, some of which are natural consequences of the discussions in Sections 5.1 and 5.2:

- The notion of *trusted* — versus untrusted — *hardware*. Most of MAFTIA's hardware is considered to be untrusted, but small parts of it are considered to be trusted in the sense of being *tamper-proof* by construction (see Section 5.3.3). Note that this notion does not necessarily imply proprietary hardware, but for example COTS hardware whose architecture and interface with the rest of the system justifies the aforementioned assumption.
- The notion of *trusted software* as a form of support component. This particular instantiation of a trusted component in MAFTIA is the way in which we endorse the notion of a fail-controlled subsystem in the runtime support. It is trusted to execute a few functions correctly (which given the scope of MAFTIA, will normally be *security-related*) albeit immersed in an environment subjected to malicious faults. The use of trusted hardware may help substantiate this assumption.
- The notion of *run-time environment*, extending operating system capabilities and hiding heterogeneity amongst host operating systems by offering a homogeneous API and framework for protocol composition. Functions supplied by the above-mentioned trusted support software are offered through the runtime API.
- Modular and multi-layered *middleware*, with a neat separation between: the multipoint network abstraction, the communication support services, and the activity support services. Despite this modularisation, the middleware is a white box, allowing users direct access to any service from any layer. A given middleware

layer may implement another instantiation of trusted MAFTIA component: a trusted distributed component that overcomes the fault severity of lower layers and provides certain functions in a trustworthy way.



AS - Authorisation Service, IDS - Intrusion Detection Service, TTP - Trusted Third Party Service

Figure 21 — MAFTIA architecture dimensions

The MAFTIA architecture can be depicted in at least three different dimensions (see Figure 21). First, there is the *hardware* dimension, which includes the host and networking devices (whose topology was briefly discussed earlier) that make up the physical distributed system. Second, within each node, there are the *local support* services provided by the operating system and the run-time platform. These may vary from host to host in a heterogeneous system, and some services may even not be available on some hosts or may have to be accessed via the network using protocols providing an appropriate degree of trust. However, at a minimum, the local services include typical operating system functionality such as the ability to run processes, send messages across the network, access local persistent storage (if it exists), etc. Third, there is the *distributed software* provided by MAFTIA: the layers of *middleware*, running on top of the run-time support mechanisms provided by each host; and MAFTIA's native *services*, depicted in the picture — authorisation, intrusion detection, and trusted third party services. Applications built to run on top of MAFTIA use the abstractions provided by the middleware and the application services to operate securely across several hosts, and/or be accessed securely by users running on remote nodes, even in the presence of malicious faults. The distributed software components of the MAFTIA architecture (middleware and services) are discussed in more detail in [Abghour et al. 2001, Abghour et al. 2002, Cachin 2001b, Cachin 2002, Dacier 2002, Neves & Veríssimo 2001, Neves & Veríssimo 2002]. In the remainder of this section, we discuss in a little more detail the hardware, the local support, and the middleware.

5.3.3 Hardware

We assume that the hardware in individual MAFTIA hosts is untrusted in general. However (see Figure 21) some hosts may have pieces of hardware that are trusted in the sense of being regarded as tamper-proof, i.e. we assume that intruders do not have direct access to the inside of the component.

Most of a host's operations run on untrusted hardware, e.g., the usual machinery of a PC or workstation, connected through the normal networking infrastructure to the Internet, which we call the *payload channel*.

Some hosts, for example, servers, will have trusted hardware components. Currently, we consider two incarnations of such hardware, both readily available as COTS components. One is a *Java Card reader*, connected to the machine's hardware, and interfaced by the operating system. The Java Card executes software functions to which an attacker does not have access and also stores keys. The other type of trusted hardware is an *appliance board with processor*. Such a board is a common accessory in the PC family that has its own resources and is interfaced by the operating system. However, an attacker does not have access to the interior of the board. The board has a network adapter to a private network, which we call a *control channel* (to differentiate it from the payload channel). An attacker does not have access to the data circulating in the control channel.

Note that contrary to the traditional security view of the term “tamper-resistance” to denote a downgraded version of “tamper-proofness”, we separate concerns between what is assumed (“tamper-proofness”) and the merit of that assumption (its coverage), which may be imperfect. For example, the Java Card is assumed in MAFTIA terminology to be tamper-proof, but this quality is trusted to the extent we believe it is worthy of that trust. This obviously depends on the level of threat we conjecture (e.g., logical vs. physical attacks, chemical attacks, etc.), and on the criticality of the functions it is expected to perform.

5.3.4 Local support

The local support dimension of the architecture (see Figure 21) consists essentially of the operating system augmented with appropriate extensions. We have adopted Java as a platform-independent and object-oriented programming environment, and thus our middleware, service and application software modules are constructed to run on the Java Virtual Machine (JVM) run-time environment. The MAFTIA run-time support also includes the APPIA protocol kernel [Miranda et al. 2001], which supports the construction of middleware protocols from the composition of micro-protocols.

The run-time support thus includes abstractions of typical local platform services such as process execution, inter-process communication, access to local persistent storage, and protocol management. These are enhanced with specialised functions provided by the Java Card based module, and the Trusted Timely Computing Base (TTCB).

The TTCB component is trusted from the viewpoints of correctness of its operation, and of intrusion prevention: the kernel provides correct security-related functions in a fault free situation, and cannot be intruded upon. In that sense, it is akin to what has been called a *security kernel* in the literature. It must follow a few construction principles that guarantee this behaviour in the face of faults:

- **Interposition**: it must by construction be interposed between the vital resources required for its correct operation and any attempt to interact with them (it is always invoked to deal with these resources)
- **Shielding**: it must be shielded (tamper-proof) from any contamination from the outside (blocks any errors propagating from the rest of the system, e.g. malicious attacks)
- **Validation**: it must be verifiable, in order to ensure very high coverage of its properties.

5.3.4.1 Java Card module

This component is used to assist the operation of a reference monitor, which supports the MAFTIA Authorisation Service (see Section 5.5.3). It plays two main roles: it checks all accesses to local objects, whether persistent or transient, and it autonomously manages all access rights for local transient objects.

The Java Card module runs partly on the operating system kernel (the reader interface part) and partly on the Java Card (the function's logic and the data structures, e.g., keys). Software components interact with it through the run-time support (the JVM). The Java Card module is used to support the operation of a local reference monitor, which runs partly on the former, partly on a JVM resident module called the dispatcher. The reference monitor controls all accesses to local objects by checking that each request carries a capability for the access. This capability may have been delivered by the authorisation server, if the access is an access to a persistent object, or by the reference monitor itself, if the access is an access to a local transient object. The Java Card is trusted to the following extent: although it is feasible to subvert it, however this requires an effort, in means and time, which makes it possible to substantiate assumptions about the resilience of a given module, or alternatively (though not currently exploited in the MAFTIA prototypes) of fault-tolerant quorums of k such replicas. In other words, the Java Card is trusted to the extent of presenting certain hardness to being broken, and of operating correctly until then.

In consequence, it is feasible to subvert a local reference monitor – in this case however, we consider that: the local damage is confined such that the global properties are not affected; and globally, no more than f local hosts may be compromised, such that the remaining hosts together enforce error confinement through appropriate fault tolerance mechanisms. This would mean that a successful attacker (i.e. an intruder) may become able to control accesses to local objects but cannot be granted access to remote objects, or impersonate a fake object for remote operations.

5.3.4.2 Trusted Timely Computing Base

The TTCB is a distributed trusted support component responsible for providing a basic set of trusted services related to time and security, to middleware protocols (communication and activity support). It aims at supporting malicious-fault tolerant protocols of any synchrony built to a fail-controlled model, such as reliable multicast, by supplying reliable failure-detection information. Furthermore, it helps to enforce timeliness specifications of protocols, even if the environment only allows this to be achieved with some uncertainty.

One important characteristic of this component is that it implements some degree of distributed trust for low-level operations. That is, protocol participants essentially exchange their messages in a world full of threats, some of them may even be malicious and cheat, but there is an oracle that correct participants can trust, and a channel that they can use to get in touch with each other, even for rare moments. Moreover, this oracle also acts as a checkpoint that malicious participants have to synchronise with, and this limits their potential for Byzantine actions (inconsistent value faults).

The other important characteristic is that the TTCB is synchronous, in the sense of having reliable clocks and being able to execute timely functions. Furthermore, the control channel provides timely (synchronous) communication among TTCB modules.

A local TTCB runs partly on the operating system kernel (the appliance board interface part), and partly on the appliance board itself. Software components interact with it through the run-time support (the JVM). The TTCB component is trusted to the following extent: it is assumed to be not feasible to subvert the TTCB, but it may be possible to interfere in its interaction with software components through the JVM. Whilst we let a local host be compromised, we make sure that it does not undermine fault-tolerant operation of the protocols amongst distributed components. Further to the TTCB tamper-proofness, we can also count on the information exchanged by the local TTCBs (including the one on the compromised host) through the control channel.

The TTCB component should be built in a way that secures both the synchronism properties mentioned earlier, and its correct behaviour vis-à-vis malicious faults, with the desired coverage. In consequence, a local TTCB would normally be built on dedicated hardware modules, with a dedicated network, as discussed earlier in Section 5.3.3. However, we also

consider simpler configurations not requiring dedicated trusted hardware for the TTCB, and study their design in order to exhibit high coverage. The software-based solution consists of a small secure real-time kernel running on the bare machine hardware, on top of which the regular operating system runs (and all the rest of the host software). The TTCB is built on the kernel, and can be trusted to the extent that this implementation enjoys the interposition, shielding and validation properties [Abrams et al. 1995]. Note that the coverage expected from this configuration cannot be worse than hardened versions of known commercial operating systems. It might actually be better, since it only addresses the inner kernel and not the operating system as a whole. It may thus constitute a very attractive implementation of MAFTIA for its cost/simplicity/resilience trade-off.

The control channel can also assume several forms exhibiting different levels of timeliness and resilience. It may or may not be based on a physically different network from the one supporting the *payload* channel. For example, virtual channels with predictable timing characteristics coexisting with essentially asynchronous channels are feasible in some current networks, even over the Internet, through QoS protocols, or through overlay networks [Schulzrinne et al. 1996]. Such virtual channels can be made secure through virtual private network (VPN) techniques, which consist of building secure cryptographic IP tunnels linking all TTCB modules together, and these techniques are now supported by standards [Kent & Atkinson 1998]. On a timeliness side, it should be observed that the bandwidth required of the control channel is bound to be much smaller than that of the payload channel. In more demanding scenarios, one may resort to alternative networks (real-time LAN, ISDN connection, GSM or UMTS Short Message Service, Low Earth Orbit satellite communication).

The TTCB is designed to act as an assistant for parts of the execution of the protocols and applications supported by the MAFTIA middleware, and consequently it can be called from any level of the middleware dimension of the architecture. The services provided by the TTCB fall into two broad categories: security-related services, and time-related services. The former include services such as trusted *block consensus*, *unilateral TTCB authentication*, and trusted *random number generation*. The latter include services such as the trusted provision of *absolute time*, *duration measurement* and *timing failure detection*. These services and the properties they guarantee are described in more detail in [Neves & Veríssimo 2002].

5.3.5 Middleware

The distribution dimension impacts on the protocol design but not on the services provided by each host. These are constructed on the functionality provided by the several middleware modules, represented in Figure 22. These interactions occur through the run-time environment. The several profiles for building protocols that were discussed earlier (e.g., time-free, timed, etc.) are achieved by composition of the micro-protocols necessary to achieve the desired quality of service. The middleware hides these distinctions from the application programmer by providing uniform APIs that are parameterised with functional and non-functional guarantees. The design of these APIs is explained in more detail in [Neves & Veríssimo 2002].

As mentioned earlier, a middleware layer may host a trusted distributed component that overcomes the fault severity of lower layers and provides certain functions in a trustworthy way. These are in turn trusted by the layers above, in a recursive way. For example, a (distributed) transactional service trusts that a (distributed) atomic multicast component ensures the typical properties (agreement and total order), regardless of the fact that the underlying environment may suffer Byzantine malicious attacks.

In Figure 22, the set of layers is divided into site and participant parts. The site part has access to and depends on a physical networking infrastructure, not represented for simplicity. The participant part offers support to local participants engaging in distributed computations. The lowest layer is the *Multipoint Network* module*, MN*, created over the physical infrastructure.

Its main properties are the provision of multipoint addressing, basic secure channels, and management communications. The MN layer hides the particularities of the underlying network to which a given site is directly attached, and is as thin as the intrinsic properties of the former allow. It also provides a run-time (JVM and APPIA) compliant interface for the protocols to be used (e.g., IP, IPSEC, SNMP).

The *Communication Support Services* module, *CS*, implements basic cryptographic primitives, Byzantine agreement, group communication with several reliability and ordering guarantees, clock synchronisation, and other core services. The CS module depends on the MN module to access the network. The *Activity Support Services* module, *AS*, implements building blocks that assist participant activity, such as replication management (e.g., state machine, voting), leader election, transactional management, authorisation, key management, and so forth. It depends on the services provided by the CS module.

The block on the left of the figure implements failure detection and membership management. These functions are performed both at site and participant level. At site level, *site failure detection* is in charge of assessing the connectivity and correctness of sites, and the Multipoint Network module depends on this information. Failure detection is not completely reliable, due to the uncertain synchrony and susceptibility to attacks of at least parts of the network. *Site membership* management, which depends on failure information, creates and modifies the membership (registered members) and the view (currently active, or non-failed, or trusted members) of sets of sites, which we call site-groups. The CS module depends on this information.

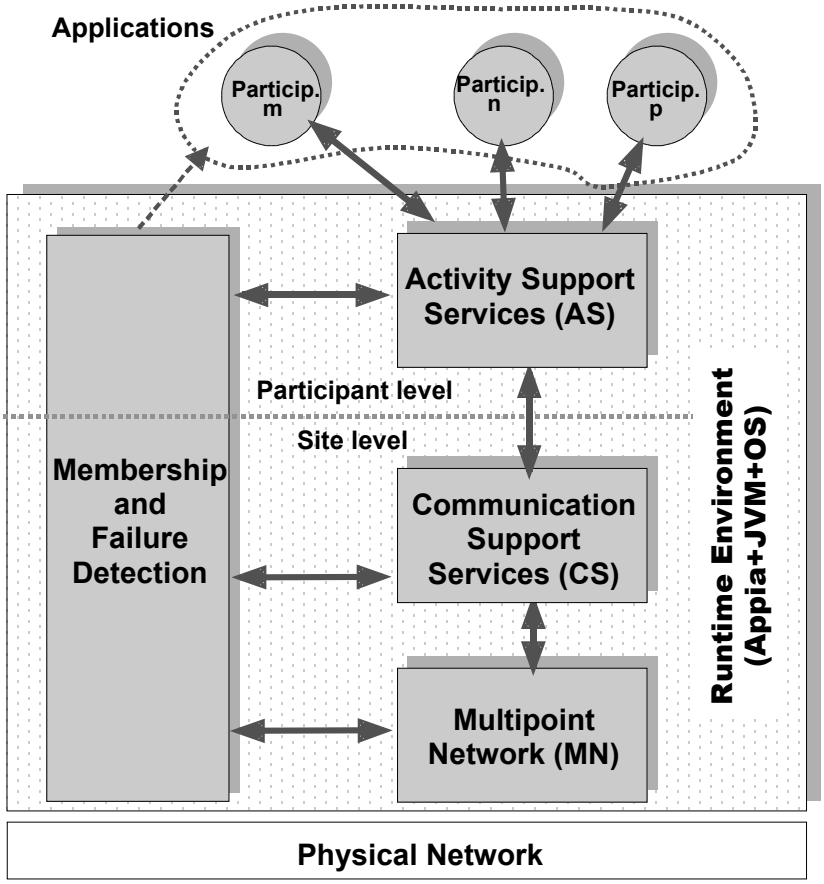


Figure 22 — Detailed architecture of the MAFTIA middleware

In the participant part, *participant failure detection* assesses the liveness of all local participants, based on local information provided by sensors in the operating system support. *Participant membership* management performs similar operations as site membership on the

membership and view of participant groups. Note that several participant groups, or simply *groups*, may exist in a single site. The separation of concerns between groups of participants (performing distributed activities), and site-groups of the sites where those participants reside (performing reliable communication on behalf of the latter) is beneficial to application structuring. This can be further enhanced by mapping more than one group onto the same site-group, in what are called *lightweight groups* [Rodrigues et al. 1996]. The Activity Support Services depend on the participant membership information.

The protocols implementing the layers described above fulfil the topology awareness property. As such, they may run differently depending on their position in the topology, although this happens transparently. For example, a site-failure detection protocol instantiated at the Facility Gateways may wish to aggregate all liveness/failure information from the sites it oversees, and gather that same information from the corresponding remote Facility Gateways. These considerations may obviously be extended to topology-aware attack diagnosis and intrusion detection.

5.4 Intrusion-tolerance strategies in MAFTIA

The goal of MAFTIA is to support the construction of dependable trustworthy applications, implemented by collections of components with varying degrees of trustworthiness. This is achieved by relying on distributed fault and intrusion-tolerance mechanisms. Given the variety of possible MAFTIA applications, several different strategies are pursued in order to achieve the above-mentioned goal. These strategies are applied at several levels of abstraction of the architecture, most importantly, in the implementation of the middleware and application services. In this section, we describe these strategies: fail-uncontrolled or arbitrary; fail-controlled with local trusted components; fail-controlled with distributed trusted components.

The conventions used for the figures in the following sections are as follows: grey means untrusted (the darker, the “less trusted”); white means trusted; the presence of a clock symbol means a synchronous environment; a crossed out clock symbol means an asynchronous environment; a warped clock symbol means a partially-synchronous environment; a key means a secure environment; dashed arrows means IPC or communication that can be interfered with; continuous arrows denote trusted paths of communication.

5.4.1.1 Fail-uncontrolled

The fail-uncontrolled or arbitrary failure strategy is based on the no-assumptions attitude discussed in Section 5.2.1. When very large coverage is sought of given mechanisms in MAFTIA, we resort to making no assumptions about time, following an asynchronous model, and we make essentially no assumptions about the faulty behaviour of either the components or the environment. Of course, for the system as a whole to provide useful service, it is necessary that at least some of the components are correct. This approach is essentially parametric: it will remain correct if a sufficient number of correct participants exist, for any hypothesised number of faulty participants *f*.

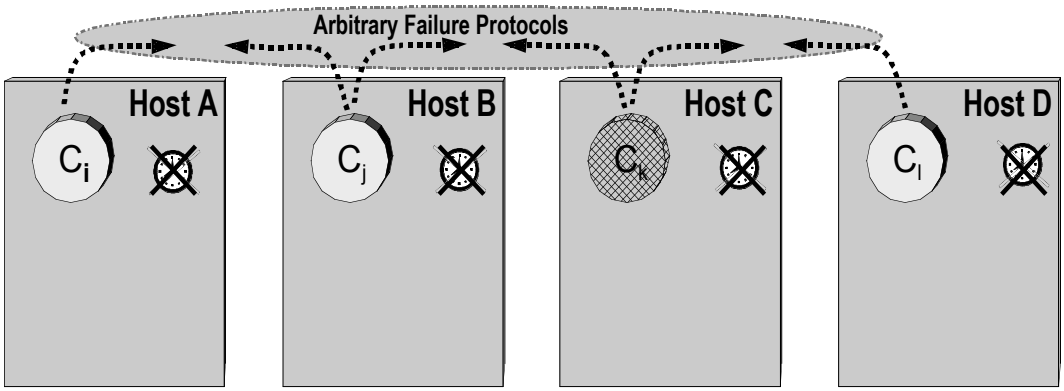


Figure 23 — Fail-uncontrolled

Figure 23 shows the principle in simple terms. The hosts and the communication environment are not trusted, and are fully asynchronous. For a protocol to be able to provide correct service, it must cope with arbitrary failures of components and the environment. For example, component Ck is malicious, but this may be because the component itself or host C have been tampered with, or because an intruder in the communication system simulates that behaviour.

Some protocols used by the MAFTIA middleware follow this strategy, in order to be resilient to arbitrary failure assumptions. They are of the probabilistic Byzantine class, and require a number of hosts $n > 3f$, for f faulty components. The MAFTIA middleware provides different qualities of service in this asynchronous profile, achieved by composition of several micro-protocols on top of basic binary Byzantine agreement, in order to achieve: reliable broadcast, atomic broadcast; multi-valued Byzantine agreement.

5.4.1.2 Fail-controlled with local trusted components

Figure 24 exemplifies a fail-controlled strategy. It consists of assuming that, as for the fail-uncontrolled strategy, hosts and communication environment are not trusted, and asynchronous. However, hosts have a local trusted component (LTC), which supports functions they can trust for certain steps of their operation. In MAFTIA, this strategy is implemented through a Java Card that equips some hosts. As such, we can construct protocols that cope with a hybrid of arbitrary and fail-silent behaviour, depending on whether a component is interacting with the other components or with the local trusted component (LTC).

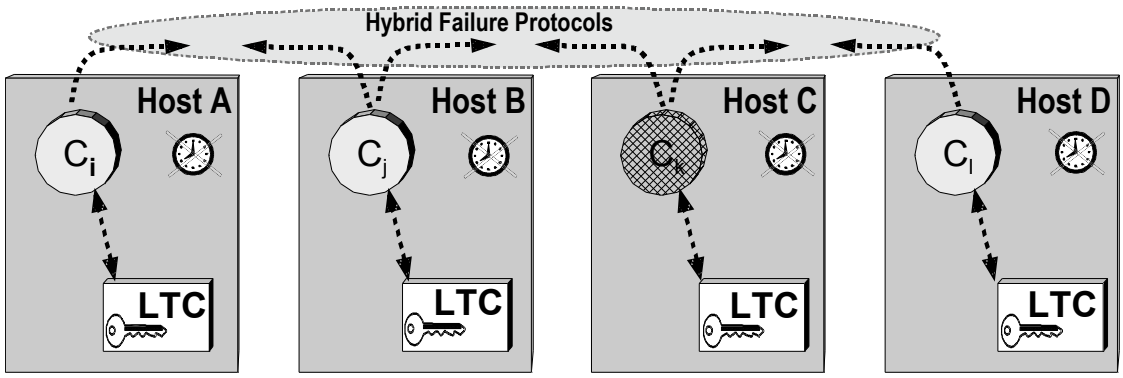


Figure 24 — Fail-controlled with local trusted components

In the example, component Ck may be arbitrarily malicious, either because the component itself or host C has been tampered with, or because an intruder in the communication system simulates that behaviour. However, unlike the fail-uncontrolled strategy, the impact of this behaviour on the other components (i.e., error propagation) may be limited, if the protocol makes components perform certain checks and validations with the LTC (for example,

signature validation), which will prevent C_k from causing certain failures in the value domain (for example, forging). An additional proviso must be made: since the host environment is untrusted, IPC between a component and its LTC may be interfered with, though in a controlled way. For example, if host B is contaminated, component C_j may behave erroneously, but protocols can be designed in a way that prevents C_j from behaving in an arbitrary (e.g. Byzantine) way towards the other hosts.

This strategy is followed in the construction of the MAFTIA authorisation service. Components run distributed fault-tolerant authorisation protocols based on capabilities that express the access control for objects. These protocols run among the authorisation server replicas and the hosts running a MAFTIA application. Given the criticality of the authorisation service, it is also worthwhile noting that the trust put on the Java Card LTC for this application is not absolute, in the sense that the higher-level protocols are ready to cope with the possibility of subversion of some Java Card modules and still ensure globally correct operation of the service.

5.4.1.3 Fail-controlled with distributed trusted components

The "fail-controlled with distributed trusted components" strategy amplifies the scope of trustworthiness of the local component support, by making it distributed. As such, certain global actions can be trusted, despite a generally malicious communication environment. This strategy is implemented in MAFTIA through the TTCB (Trusted Timely Computing Base), which builds trust on global (distributed) time-related and security-related properties (such as global time, distributed durations, block agreement). One main impact of relying on the TTCB is that timed behaviour can be supported globally in an intrusion-resilient way, as suggested by the warped clocks in Figure 25: the system is assumed to be *partially synchronous*, that is, anywhere in the interval ranging from time-free to fully synchronous, depending on the environment. This strategy assumes, as for the preceding strategies, that the hosts and communication environment are not trusted.

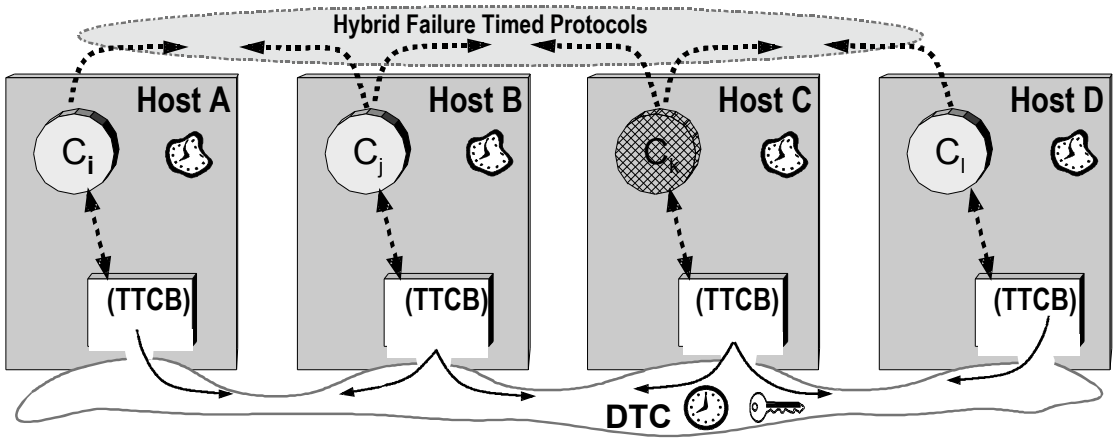


Figure 25 — Fail-controlled with distributed trusted components

The distributed trusted component (DTC) is implemented by the local TTCBs interconnected by a control network. As with the "fail-controlled with local trusted components" strategy, in order for a protocol to be able to provide useful service, it has to cope with a hybrid of arbitrary and fail-silent behaviour, depending on whether a component is interacting with the other components or with the TTCB. Consider the example of Figure 25, where again component C_k or host C may be arbitrarily malicious. Like the "fail-controlled with local trusted components" strategy, the impact of the faulty behaviour of these components may be limited by enforcing certain validations with the local TTCB. However, the fact that the TTCBs are interconnected and can exchange information and perform agreement in a secure way — through the control channel — further limits the potential damage of malicious behaviour: the DTC 'knows' directly what each of the components in different hosts 'say',

unlike the solution with LTCs, where an LTC only 'knows' what a remote component 'says', through the local component. To achieve this, the TTCB allows the set-up of secure channels with any local component, and offers a low-level block consensus primitive. For example, components C_i through C_l could set up secure IPC with the TTCB, through which they would run such a consensus as part of the execution of some protocol.

The other relevant aspect of the TTCB strategy is time. The TTCB supports timed behaviour in an intrusion-resilient way. As discussed in Section 5.2.6, timed systems are fragile in that timing assumptions can be manipulated by intruders. The TTCB supplies constructs that enable protocols to tolerate this class of intrusions. These are obviously related to the trusted time-related services briefly described earlier, namely absolute time, duration measurement and timing failure detection. As suggested in Figure 25, the TTCB DTC is a fully synchronous subsystem. It supplies its services to the payload system, which can have any degree of synchronism, as suggested by the warped clock. The TTCB does not make the payload system "more synchronous", but allows it to take advantage of its possible synchronism, in the presence of faults, both accidental and malicious. As such, the TTCB can assist an application running on the payload system to determine useful facts about time: for example, be sure it executed something on time; measure a duration; determine it was late doing something, etc. Then, the payload system, despite being imperfect (it suffers timing faults, some of which may result from attacks), can react (implement fault-tolerance mechanisms) based on reliable information about the presence or absence of errors (provided by the TTCB at its interface).

Depending on the type of application, it is not necessary that all sites have a local TTCB. Consider the development of a fault-tolerant TTP (Trusted Third Party) based on a group of replicas that collectively ensure the correct behaviour of the TTP service vis-à-vis malicious faults. The nodes hosting these replicas have TTCBs that support the execution of the group communication and replica management protocols under a timed model.

Several of the MAFTIA middleware protocols follow the "fail-controlled with TTCB" strategy. These protocols are group-oriented, deterministic, and can provide timeliness guarantees. The MAFTIA middleware provides different qualities of service in this timed profile by composing several micro-protocols on top of basic unreliable multicast. For example, this is the way in which reliable multicast and atomic multicast protocols are achieved.

5.5 Examples of MAFTIA intrusion tolerant services

To illustrate the application of MAFTIA intrusion-tolerance strategies to the problem of building intrusion tolerant trusted services, we briefly discuss four examples that are being developed within the project, namely intrusion-detection service, trusted third party services, authorisation service, and transaction service. More details about these services can found in [Abghour et al. 2001, Abghour et al. 2002, Cachin 2001b, Cachin 2002, Dacier 2002, Neves & Veríssimo 2001].

5.5.1 Intrusion-detection service

The goal of MAFTIA is to support the construction of dependable trustworthy applications by distributing trust. As discussed in Chapter 4, intrusion detection is relevant at all levels of the architecture. For example, the operating systems used by the MAFTIA platform should have integrity checking and configuration checking enabled. Reports of attacks staged against servers running on the platforms should be noted. Periodic auditing or review of the systems and their administrators should be performed. The logging information generated by the MAFTIA middleware, support structures, and so forth may also be used to support intrusion detection. For example, repeated incorrect calculations or evidence of a dictionary attack against cryptographic mechanisms should be noted.

Not only should the intrusion-detection service rely on information collected from every layer of the architecture, but also the intrusion-detection service should itself be intrusion tolerant. Sophisticated attackers are likely to target the intrusion-detection system in an attempt to disable it or in order to disguise their subsequent attacks. The strategies described in section 5.4, middleware services such as secure channels, and the principle of error compensation can all be used to make the intrusion-detection service intrusion tolerant.

The choice between the "fail-uncontrolled" and the "fail-controlled with distributed trusted components" strategies for the design of the intrusion-detection components depends on factors such as the number of components required and where they are placed. Some components will be able to use specialised platforms that support TTCBs, for example, standalone network-based sensors. Other components may have to co-exist with applications on standard platforms and will have to adopt a fail-uncontrolled strategy.

The question of whether the intrusion-detection system should use the reliable and secure communication channels provided by MAFTIA is answered by consideration of the failure-modes. Naturally, one would not wish to use a communication channel to signal failure of the communication channel itself. In addition, one would not wish to invoke a large distributed architecture to communicate between two components within a single trust domain. In intrusion-detection system settings where error compensation does not make sense, we can use much simpler mechanisms and channels (as described in Section 4.2.4).

Error compensation could be used to improve the robustness of the communication channels that the intrusion-detection components use to communicate. Error compensation relies upon the erroneous state containing enough redundancy to enable its transformation into an error-free state. In intrusion-detection system settings where error compensation is appropriate, we can benefit by incorporating redundancy into the data sent through the communications channels. Message selection algorithms can be applied to the messages received over multiple channels. This would enable faults due to intrusion or other causes to be masked.

These architectural trade-offs in building an intrusion tolerant intrusion-detection system are explored further in Chapter 6 of [Dacier 2002].

5.5.2 Distributed trusted services

These services are based on the fail-uncontrolled strategy, and error compensation. Error compensation is implemented by using active or "state machine" replication [Powell et al. 1988] in the Byzantine model. The general idea is to implement a server providing the service as a deterministic state machine and replicate it. We assume a static server group of n replicated servers, of which up to t may fail in completely arbitrary ways. Clients send their requests to the server group, the replies are collected by the client and a selection algorithm is applied to determine the correct reply. This allows the corruption of a subset of the servers to be tolerated. Requests to the services are delivered by the broadcast protocols described in [Cachin 2001b] that have been designed to cope with arbitrary failures of components and the environment. A broadcast is started when the client sends a message containing the request to a sufficient number of servers. In general, the client must send the request to more than t servers or a corrupt server could simply ignore the message; alternatively, one could postulate that one server acts as a gateway to relay the request to all servers and leave it to the client to resend its message if it receives no answer within the expected time.

Depending on whether it is necessary to maintain causality among client requests, a service may use atomic broadcast directly or secure causal atomic broadcast otherwise. If the client requests commute, reliable broadcast suffices.

Each server returns a partial answer to the client, who must wait for at least $2t+1$ values before determining the proper answer by majority vote. Since atomic broadcast guarantees that all servers process the same sequence of requests, the client will obtain the same answer

from all honest servers. If the application returns a digital signature, the answers may contain signature shares from which the client can recover a threshold signature.

The following are examples of the types of applications envisaged as being made intrusion tolerant using this approach:

- *Digital Notary Service*. A number of applications require a single counter to be provided by a trusted central authority. In its most basic form, a digital notary service receives documents, assigns a sequence number to them and certifies this by its signature.
- *Fair Exchange TTPs.* Fair Exchange protocols are useful in electronic commerce for digital content selling, certified email or electronic contract signing. The fairness property ensures that either both parties that wish to exchange items get the item they are supposed to, or that neither party gets anything.
- *Certification Authority* (CA). A CA is a service run by a trusted organisation that verifies and confirms the validity of a public key. The issued certificate usually also confirms that the real-world user defined in the certificate is in control of the corresponding private key. The CA links the public key to a user's identity by signing the two together under the CA's private signing key.
- *Authentication Service*. The basic task of an authentication service is to verify the claimed identity of a user or a process acting on behalf of a user. This service is used when privileges are granted according to user identity (e.g., by an authorisation service), or when the authentic identity of a user must be recorded for accountability.
- *Authorisation Service*. An authorisation service is in charge of granting or denying rights for specified subjects to carry out specified operations on specified objects. MAFTIA is developing a distributed trusted authorisation service for multiparty transactions that is sketched out in the following sub-section.

[Cachin 2001b] discusses this approach to building dependable trusted third party services in more detail.

5.5.3 Authorisation service

Most current Internet applications do not use authorisation services. Such applications are based on the client-server model where, typically, the server distrusts clients, and grants each client access rights according to the client's identity. Moreover, the server must usually record the client's identity and as much information as possible on the transaction to support dispute resolution. It is then easy to correlate such personal information for marketing purposes: the client's identity, usual IP address, postal address, credit card number, purchase habits, etc. Such a model is thus necessarily privacy-intrusive.

Furthermore, the client-server model is not rich enough to cope with complex transactions involving more than two participants. For example, an electronic commerce transaction requires usually the cooperation of a customer, a merchant, a credit card company, a bank, a delivery company, etc. Each of these participants has different interests, and thus distrusts the other participants.

Authorisation services have been introduced in locally distributed systems, mainly to facilitate security management (Delta-4 [Blain & Deswarte 1990], HP Praesidium authorisation server, ADAGE [Zurko et al. 1999]). In these cases, according to a security policy, the authorisation service distributes authorisation tickets or capabilities, which are later presented as proofs that an operation has to be granted by another server. The authorisation service usually uses an authentication service and locally stored information to decide whether or not to authorize a given operation to a given user.

Within the MAFTIA project, we are developing authorisation schemes that can grant fair rights to each participant of a multiparty transaction, while distributing to each one only the information strictly needed to execute its own task, i.e., a proof that the task has to be executed and the parameters needed for this execution, without unnecessary information such as participant identities. These schemes are based on two levels of protection:

- An *authorisation server* is in charge of granting or denying rights for high-level operations involving several participants; if a high-level operation is authorized, the authorisation server distributes capabilities for all the elementary operations that are needed to carry it out.
- On each participating host, a *reference monitor* is responsible for fine-grain authorisation, i.e., for controlling the access to all local resources and objects according to the capabilities that accompany each request. To enforce hack-proofing of such components on off-the-shelf computers connected to the Internet, critical parts of the reference monitor will be implemented on a trusted component based on a Java Card.

The implementation of an intrusion-tolerant authorisation service relies on applying error compensation, and the "fail-controlled with local trusted components" strategy. Error compensation is implemented through the combined use of active replication and fragmentation-redundancy-scattering [Deswarte et al. 1991]. The authorisation service is composed of replicated and diverse servers, operated by independent people, so that any single fault or intrusion can be tolerated without degrading the service. Confidential authorisation data is fragmented, replicated and scattered across the servers. In order to reconstruct the data multiple servers must co-operate. This means that as long as only a minority of the replicas are compromised there is no loss of confidentiality of authorisation data. A "fail-controlled with local trusted components" strategy is used, based on threshold-signature algorithms. Access to application resources is controlled by the local trusted component. If the latter is compromised then the effect of the failure is localised due to the limited trust put on the Java Card: the corruption of the local host gives no privilege to access remote objects, and a corrupt host cannot impersonate another host.

For more details of the Authorisation Service, see [Abghour et al. 2001, Abghour et al. 2002] and two recent publications [Deswarte et al. 2001, Deswarte et al. 2002].

5.5.4 Transaction service

A transaction is a set of requests that have the *ACID* properties [Härder & Reuter 1983]: atomicity, consistency, isolation and durability. Atomicity is the property that a transaction must be all or nothing. Consistency is the property that a transaction takes the system from one consistent state to another consistent state. Isolation is the property that the intermediate effects of a transaction must not be visible to another transaction. Durability is the property that the effects of a transaction are permanent.

Typical transaction service architectures are composed of clients, resource managers and transaction managers. Clients interact with the transaction manager to establish transactions. Within the scope of a transaction, the clients operate on resources via resource managers. A resource manager is a wrapper for resources that allows resources to participate in two-phase commit [Gray 1978] and recovery protocols coordinated by a transaction manager, and controls the access that clients have to resources. The transaction manager is primarily a protocol engine. It implements the two-phase commit protocol and recovery protocol.

The MAFTIA transaction service supports multiparty transactions and provides atomicity in the face of failure due to intrusions as well as crash failure. Multiparty transaction support allows one client to begin a transaction and to invite other clients to join with it in the transaction context. All clients within the transaction context can access transactional resources in a cooperative manner using application-specific protocols while competing for

access to resources with clients who are not within the same transaction context. The MAFTIA transaction service preserves atomicity in the face of failure due to intrusions as well as hardware or software failure. It achieves this by applying error compensation and the strategies: “fail-controlled with local trusted components” and “fail-controlled with distributed trusted components”.

Error compensation is implemented using active or “state machine” replication [Powell et al. 1988]. The transaction service is composed of replicated and diverse resource manager and transaction manager servers. We rely upon the MAFTIA middleware’s communication services to implement the replication. Therefore, in order for the transaction service to tolerate intrusions, we need the communication services to be intrusion tolerant.

Two different strategies can be used to make the communication services intrusion tolerant. The “fail-uncontrolled” strategy can be used to provide fault-tolerant atomic broadcast for systems where Byzantine behaviour by users is possible and we cannot make timing assumptions. The fault-tolerance provided by this strategy depends upon the use of time-free probabilistic Byzantine protocols. The “fail-controlled with distributed trusted components” strategy can be used to provide fault-tolerant atomic broadcast where a TTCB is present. The tamper-proof construction of the local TTCB and the control channel prevents the host engaging in Byzantine behaviour or being vulnerable to timing attacks.

We must also prevent unauthorised clients from interacting with the transaction service. The “fail-controlled with local trusted components” strategy provides authorisation for the users of the transaction service. Capabilities for accessing resources are issued by the distributed authorisation server, and checked by the local trusted component. In addition, since the component is tamper-proof, private keys that could be used by an adversary to gain access to remote resources will not be revealed even if the host is compromised. For example, compromising the transaction manager will not result in the adversary gaining control of the resource managers.

The design of the Transaction Service is described in more detail in [Neves & Veríssimo 2001].

Chapter 6 Verification and Assessment

The purpose of verification and assessment in secure systems is two-fold: to uncover design faults, i.e., human-made development faults, which are typically accidental; and to provide positive evidence of the integrity of the system under scrutiny. On the one hand, verification and assessment is a security method in itself, a part of vulnerability removal. On the other hand, it can be seen as an assurance technique accompanying, and orthogonal to, many other security methods, ensuring that they achieve their objectives. This second view corresponds to the duality between functional and assurance requirements in security evaluation criteria, such as [DIS 15408-1-3].

The verification and assessment work-package within MAFTIA employs non-automated proof and automated proof. The non-automated proof is largely based on the *rigorous secure reactive systems theory* of Pfitzmann and Waidner [Pfitzmann & Waidner 2001]. The automated proof takes the form of model-checking, where the models and specifications are described in the process algebra CSP [Hoare 1985, Roscoe 1998], and the model-checker FDR [Formal Systems (Europe) Ltd] is used to help reason about them.

6.1 Special purpose of verification and assessment in MAFTIA

A discussion covering all aspects of assessment and verification for such general topics as considered in MAFTIA would be completely beyond the scope of this document. We therefore concentrate on aspects where new developments were needed for MAFTIA. We had two main goals here:

1. To provide a rigorous formalisation of the basic concepts of MAFTIA, in particular as presented in Chapters 2 and 3.
2. To develop new specification and verification techniques in areas where traditionally separate sub fields of dependability meet and no appropriate techniques exist yet.

These aspects are described in Sections 6.2 and 6.5. In Section 6.4, we give a brief overview of the general role of verification and assessment in dependability from a MAFTIA perspective.

6.2 Formalisation of basic concepts of MAFTIA and architectural principles

Chapters 2 and 3 of this report define the basic MAFTIA concepts in a rather precise way, but entirely in natural language. In particular, there are general *system terms* like “component” and “specification”, relative to which *dependability-specific terms* such as “fault”, “error”, “failure”, and the various classes of security methods are defined. For use in verification, all these “meta-definitions” must be cast into mathematically rigorous concepts. They will thus gain in precision, but also lose in generality. For instance, all the rigorous models we have used in MAFTIA are discrete, although the meta-definition that a maliciously faulty component behaves “arbitrarily” could certainly also be expressed using a continuous system model.

For the system terms, one might have hoped to reuse one of the many existing general system models, so that only new definitions for dependability-specific terms would be needed. However, the scope of MAFTIA includes cryptographic subsystems, and so this was not possible, as we are unaware of any fully defined system model that includes all the necessary aspects, such as probabilism, resource limitations, and restricted adversarial scheduling of events.

The current formalisation is mainly presented in [Adelsbach & Pfitzmann 2001] and [Adelsbach & Steiner 2002]. In the following, we give a guide to it alongside the meta-definitions of the earlier chapters of the present document; the terms from the meta-definitions are in bold face.

6.2.1 Behaviour and structure of a system

Formalisation of the basic concepts of MAFTIA begins with formalising the notion of a system: **Atomic systems** (entities, components) are formalised as probabilistic I/O automata[16] called *machines*. Their **states** are in general just a set, which could be implemented as the current value of a tuple of variables. Atomic systems have several *ports* for interaction with their **environment**. They can be composed into larger systems by linking their ports; concretely this is done by a naming convention. This is the **structural** point of view. Some ports may remain free in this composition, so that one may get a larger system that can again interact with its environment. One can combine several machines into a larger machine; this gives the recursive view of **systems** as **components** of other systems.

The most general definition of **behaviour** is made for such collections of atomic systems; it is a probability distribution on possible *runs*, i.e., on sequences of states, inputs and outputs. There is no specific definition of "the **service**" delivered by a system, but one could define it as the I/O behaviour of the system combined into one machine, or even as a description of this without states at all as in [Gray III 1992]. Similarly, the general model does not define the notion of "service element" (although many concrete systems do offer them, e.g., as reactions on different classes of inputs), because typically the reactions are interlinked via global parts of the state, i.e., a service element cannot formally be defined in isolation.

The behaviour definition, by its nature, must include a model of time, a concept informally discussed in Section 5.2.6. We have defined a synchronous and an asynchronous model, both already including the fact that malicious components may want to deviate from timing requirements.[17] In previous work, we reported on the CSP modelling of synchronous [Adelsbach & Pfitzmann 2001], asynchronous [Creese & Simmonds 2002], and hybrid synchrony [Adelsbach & Creese 2003] models. This modelling was partly general theory and partly by example through the modelling of specific MAFTIA protocols. For that modelling we endeavoured to capture the formalisations of those synchrony models as accurately as possible.

Specifications occur in several forms in the model. First, for atomic systems (components), the desired state-transition function can be considered to be a specification in itself. Then any deviation, even in the internal state, is considered to be not only an error but also a failure. This is also the view that one takes if one regards components to be atomic from the dependability point of view, i.e., with no internal fault-tolerance measures. Secondly, there are different classes of specifications that leave more freedom, in particular for internal fault-tolerance. We have formalised certain important classes, but as these are mainly a formalisation of the concepts described in Section 3.1, we postpone further discussion of them until 6.2.3.

6.2.2 Modelling faults

Due to the mathematical nature of a formalisation, we mainly model errors and failures, and not faults ("causes").

---

[16] Note that our formulation of the I/O automata model is not the traditional one, e.g. DFA or NDFA, but is computationally equivalent to Turing Machines.

[17] The asynchronous model allows the derivation of (partially) timed models as specializations by introducing specific scheduler components [Backes 2002].

In principle, faults may cause a system to behave unexpectedly. Verifying the dependability of concrete systems, however, clearly presupposes **failure assumptions** as in Section 5.2.1. Under a particular fault model, the behaviour of the system is expected to meet its specification. The verification is only meaningful in practice if the failure assumptions are correct, because correctness is only checked for with respect to the fault model.[18] Model checking is an ideal choice for formulating such testing, as it provides one with details of system behaviour.

The formalisation captures arbitrary failure assumptions by representing a system as an arbitrary set of possible machine collections, where each machine is defined to fail according to its failure assumptions. However, it also provides various predefined specialisations corresponding to particular sets of **common failure assumptions**. These include arbitrary and controlled failures of individual components, and passive or active tapping of communication links. In particular, the failure model replaces all maliciously failed components by one component '*A'* the *adversary*, so called because the malicious behaviour may be co-ordinated. To allow computationally secure systems, a runtime restriction may be made on *A* (this is equivalent to implementing a controlled failure assumption, cf. Section 5.2.1). In addition, there is still a model of an arbitrary honest **user** '*H'* to whom a service is guaranteed. We also have a predefined specialisation to dynamic failures (see [Adelsbach & Creese 2003] for a detailed example).[19]

Hybrid failure assumptions assume parts of the overall system to exhibit a fail-controlled behaviour while other parts of the system are assumed to fail in an uncontrolled way. As discussed in Section 5.2.3, we have to achieve coverage of controlled failure assumptions by augmenting fail-uncontrolled components by additional enforcement components that perform attack/intrusion prevention/masking and by proving that the controlled failure assumptions hold. Modelling such composed fail-controlled components as well as the overall system is straightforward, using the formalisation of basic concepts as outlined above.

Furthermore, our composition theorems (see Section 2.8 of [Adelsbach & Pfitzmann 2001], and Section 2.5 of [Adelsbach & Steiner 2002]) allow us to prove the correctness and security of such systems in a modular way: We first prove that the system's sub-components, possibly consisting of several fail-uncontrolled sub-components and enforcement components, fulfil their specification[20]. Then we can prove the security of the overall system by assuming that all sub-components fulfil their specification, i.e., by substituting all sub-components by their specification. Finally, the composition theorem guarantees that substituting the specification of sub-components by their implementation preserves the proven properties of the overall system.

As in the non-automated formalism, described above, in our CSP modeling we also tended to replace all malicious components by a single adversary process, at least when the corruption model was assumed to be **static**, which for all our protocol verifications was the case. However, in CSP the modelling of faults is, for all practical purposes, restrained by state-space considerations. The CSP models not only have to be finite-state, but also the state has to be relatively small in order for the models to be tractable to automated checkers. This means that in practice we cannot model the generalised adversary as presented in the non-automated formalism. In particular, probabilistic behaviour is generally not possible to model tractably, and thus, for example, imperfections in cryptographic primitives are usually abstracted away

---

[18] Of course, it is also possible to test the sensitivity of a system to faults outside its failure assumptions.

[19] Including dynamic **repairs** should be fairly easy if repaired components restart in a fixed state and are brought up to date within the system. If an appropriate state must be set by hand in the repair, this is harder to model.

[20] Here, a component's specification comprises its service as well as the assumed fail-controlled behaviour.

by Dolev-Yao type assumptions in CSP [Dolev & Yao 1983]. In CSP it is usually necessary to model adversaries particular to the service under consideration according to the failure and trust assumptions of that service. This is exemplified by the CSP modelling of particular MAFTIA protocols as reported on in [Adelsbach & Pfitzmann 2001], [Creese & Simmonds 2002] and [Adelsbach & Creese 2003]. In particular, [Creese & Simmonds 2002] discusses how dynamic and stop failures can be modelled in CSP.

6.2.3 Specifications for dependability

As discussed in Section 2.2, dependability in general just means that a system reliably **delivers** a **desired service**. Hence there is, *a priori,* nothing specific about specifications for dependability — anything we may want to specify we may also want to have delivered reliably. Thus, to a large extent, dependable *fulfilment* of a specification is simply defined as follows: all sets of possible machine collections that are possible under the failure assumption (see Section 6.2.2) fulfil a normal specification in the normal sense.

Nevertheless, there are certain dependability-specific aspects of specifications that are important, in particular security-specific ones.

The first aspect is an inclusion of **confidentiality** properties. There are two approaches to this:

1. One takes a "normal" specification that describes a service unambiguously (i.e., without remaining degrees of freedom) and extends the formal notion of fulfilment to the fact that an adversary on the real system should not learn more than an adversary on the specification. This is also called *simulation*.
2. One defines specific *confidentiality properties*, e.g., that an adversary cannot gain any knowledge about certain inputs via the system.

The second aspect is *fulfilment in a computational sense*. This becomes necessary for almost all systems containing cryptography, because most cryptographic systems are easily breakable given arbitrary computational resources. In cryptography this aspect is traditionally put into the specification, but we claim that in large-scale systems the specifications should stay "normal", i.e., not clogged up with such details, and the imperfections should be defined as a specific semantics.

MAFTIA work on verification and assessment has in particular extended the first approach to including confidentiality properties, and has also for the first time clearly defined fulfilment in a computational sense for both simulations and individual integrity properties.

The division into **availability**, **integrity** and **confidentiality** occurs in the general rigorous model, but as classes of properties rather than "attributes": integrity properties (where the "validity" of the data can be defined in terms of consistency of the state of the target system) are equivalent to safety properties in the sense of [Alpern & Schneider 1985]; availability corresponds to liveness properties; and confidentiality corresponds to non-interference properties. Simulatability definitions cover all three classes.[21] A similar unification of all three security properties in terms on non-interference is presented in [Focardi & Martinelli 1999].

Readers interested in seeing how **authenticity** and **non-repudiation** can be considered as integrity properties, as claimed in Section 3.1.3.4, are referred to the example in Chapter 4 of [Adelsbach & Pfitzmann 2001].

The CSP modelling performed here we treat **confidentially** properties as safety properties that can be checked for by showing that certain undesirable events do not happen in any behaviour of the system. Strictly speaking, confidentiality is neither a trace not a liveness property of the

[21] Security **attributes** are interesting for specific system classes with clearly defined "data items" or messages, to which one can attach these attributes. An attempt to use such attributes consistently in a fairly large architecture was made in SEMPER [Asokan et al. 2000].

target system and should be modelled in terms of process equivalence as discussed in Chapter 3. More details can be found in. for example, [Ryan 2000]. Treating confidentiality as a safety property does provide a reasonable approximation and is far more tractable from a modelling point of view than treating it as a fully blown non-interference property, as discussed in more detail in section 6.3.3.

Likewise, our verification of availability or liveness properties posed no significant theoretical problems for those cases where the service was assumed to be running in an environment with timeliness guarantees - for example, that messages are guaranteed to be delivered in a certain period of time. However, there are several important asynchronous distributed MAFTIA services that assume a notion of **eventual delivery** – i.e. that messages "eventually" get through after a indeterminate period of time – in order to provide availability guarantees. Verifying such availability guarantees is far more difficult. A general methodology for doing this in CSP was detailed in Chapter 3 of [Creese & Simmonds 2002].

6.3 Security models

We will regard a security model as a mathematical framework in which security properties, policies, designs and mechanisms can be precisely formulated and analysed. It will thus be an abstract model of the security relevant aspects of computation and interaction. Determining which aspects of a system's behaviour are security relevant is delicate and error prone.

Historically there have been two main schools of security models: the access control school, exemplified by the Bell-LaPadula model, and the information flow school, exemplified by the Goguen-Meseguer notion of non-interference. These are often regarded as competing and conflicting and a number of secure system developments have followed one school or the other. In fact both have a role to play and it is important to understand the relationship between them. First we present a very brief overview of these styles of model. For more detail we refer the reader to [Gollmann 1999, McLean 1994, Ryan 2000, Samarati & Vimercati 2000].

6.3.1 Access control models

The first, shall we say, quasi-formal model of computer security was that of Bell-LaPadula [Bell & LaPadula 1974]. For the purposes of illustration we describe a simplified version of the model. The key ingredients are:

- A lattice L of security classifications/clearances,
- A set of subjects S and a set of objects O,
- A pair of operations, "read" and "write",
- A mapping $clear: S \rightarrow L$ and a mapping $class: O \rightarrow L$.

A Mandatory Access Control (MAC) Multi-Level Security (MLS) policy in encoded in this model by a pair of rules:

- No read up (simple security property): subject $s \in S$ can read object $o \in O$ if and only if $clear(s) \geq class(o)$,
- No write down (*-property): subject $s \in S$ can write to object $o \in O$ if and only if $class(o) \geq clear(s)$,

where the $\geq$ relation is that defined by the lattice. The intuition behind this is that a subject, typically a person, should only be allowed to read an object, a file say, if their clearance is at least equal to the classification of the file. The classification of an object is intended to represent the level of sensitivity of the information it contains. The clearance of a subject reflects the level of trust bestowed on them. Suppose that we have a simple (linear) lattice:

Top secret

|

Secret

|

Unclassified

Then a subject with secret clearance may have read access to secret and unclassified data (secret dominates secret and unclassified) but not top secret. Note that the lattice relation is reflexive, so points dominate themselves.

The so-called *-property is intended to ensure that information cannot be written down the lattice. The *-property was actually added in a later version to counter the observation that the simple security property alone would allow Trojan horses: code inserted at the high level capable of signalling information down through the lattice. If we assume that the read and write operations are the only means for information to flow within the system and constitute one way flows in the intuitive sense then we see that these rules together prevent any downward flow of information.

More generally, an Access Control Policy (ACP) is a statement of the rules constraining which privileges are allowed to which subjects, i.e., who may have what accesses to which system resources under what circumstances. The policy will also lay out the rules constraining how the allocation of privileges may evolve, for example, how they may be delegated by one subject to another.

In order to specify and model a general ACP, we need to set up a simple, abstract model of the system. This will comprise a set of subjects S, which can be thought of as active entities, e.g., users or their proxies, and a set of objects O, passive processes, e.g., resources (files, printers, CPU, etc.). We also need a set of privileges P, that can be thought of as access rights, e.g., read, write, execute, delegate and so on. A simple, static ACP can now be expressed as a mapping

$$\Phi: S \times O \times P \rightarrow B$$

i.e., a mapping from the triples formed from subjects, objects and privileges to the Booleans. Thus, given a request for subject *S* to exercise privilege (access) *P* over Object *O*, the ACP returns *True* or *False*.

Such an ACP can equivalently be encoded as an Access Control Matrix (ACM), with subjects in the rows, objects in the columns and privileges in the entries. We can now represent the (abstract) security state of the system as matrix whose rows are the subjects and whose columns are the objects and whose entries are privileges.

More generally it may be necessary to encode more information into the security state. For example, if the policy involves rules that depend on historical information (e.g. previous accesses), location or time, then such information will have to be included in the security state. Thus the security state will be constructed so as to ensure that it embodies sufficient information for any access control policy decision to be determined.

An ACP can be expressed as a trace property, i.e. a set of sequences of legal accesses. An ACP thus embodies the *enforceable* aspects of the security policy [Schneider 2000], where an enforceable policy is one that could be enforced by an *execution monitor*. See the discussion in Section 3.1.3.5.

Formulating the execution monitor concept in CSP is particularly appealing as we can use CSP parallel composition to good effect. The CSP parallel operator corresponds to a rendezvous abstraction of process interaction. Processes have to synchronise on actions they have in common, i.e. they must execute in lock-step.

Given an arbitrary system S and trace property φ with corresponding execution monitor EM_{φ}, the parallel composition $S||EM_{\varphi}$ will automatically satisfy φ. In other words, a given trace property can be enforced by composing a suitable execution monitor in parallel with it.

6.3.2 Information flow models

A number of shortcomings were quickly recognised with the Bell-LaPadula style of model. Firstly it turns out to be very difficult to identify all information channels in a real system. Furthermore, many of the terms employed are not given precise let alone formal definitions. In particular the terms “read” and “write” depend on our intuitive understanding of these terms. These difficulties quickly found concrete manifestations when it was discovered that implementations consistent with the Bell-LaPadula model had so called covert channels. These are implicit information channels not explicitly identified in the model.

Examples of how covert channels might arise stem from the implicit assumption that read and write operations are one-way. Thus we assume that writing to a file involves only a transfer of information from the subject to the object, the file. In fact, if we examine the way a write operation is implemented, we find that there will typically be the possibility of information flow in the other direction. An edit command might be refused or delayed, for example, and this could serve to signal from a high object (file) to a low subject (user). In particular the system might be implemented in such a way that a request from an unclassified user to create a file that already exists at secret level is rejected. This would provide a means for a secret user (or Trojan horse installed at the secret level) to signal to the unclassified user: creating a secret file of an agreed name could convey one bit of information.

This particular problem is easily sidestepped by implementing a file system that uses different name spaces for files of different classification levels, thus allowing unclassified and secret files with the same name to co-exist, unaware of each other’s existence. Now, if the unclassified user requests the creation of a new file that happens to have the same name as an existing secret file, the file will be created and no information about the existence of the secret file conveyed. The point is that the flawed implementation is perfectly consistent with the Bell-LaPadula model.

These considerations led to the proposal of a radically different way of formulating confidentiality properties, the so-called *non-interference* approach. This is commonly attributed to Goguen and Meseguer [Goguen & Meseguer 1982] but the basic ideas go back to Feiertag and Cohen [Cohen 1977, Feiertag 1980]. The basic intuition of very simple: we want to formalise the idea that information cannot flow via a system *Sys* from one process, High say, to another, Low say. We assume that High and Low have disjoint interfaces with *Sys* and that any interactions they may have must be mediated by *Sys*. We assert that High is non-interfering with Low via *Sys* if it is never possible for a change in High’s interaction with *Sys* to produce an observable change in Low’s interactions with *Sys*. Put differently: Low’s interactions with *Sys* are independent of High’s. This gives us a way of formalising the idea that no causal influence can flow from High to Low via *Sys*.

Giving this deceptively simple intuition a precise formulation turns out to be surprisingly delicate in the presence of non-determinism. The details are unimportant here, a process algebraic formulation can be found in [Ryan 2000] for example. What is significant, from the point of view of this document, is that the property of being non-interfering is not a trace property. It cannot be formulated as a predicate over traces or, essentially equivalently, over states. In process algebraic terms, non-interference can be cast as an assertion that different

instantiations of the system (corresponding to different High behaviours) are indistinguishable to Low.

This can be expressed formally: if S is non-interfering from High to Low:

$$\forall\, t, t' \in \tau(S) \quad t \sim_L t' \Rightarrow A_L(S/t) \cong A_L(S/t')$$

$\tau(S)$ denotes the set of traces of S. $\sim_L$ is an equivalence relation over traces. Usually two traces will be deemed equivalent if their projections onto the Low events are equal, but more general equivalences are possible. A_L denotes an appropriate abstraction operator that yields the Low level view of the system. S/t denotes the system S after it has executed the trace t. Finally $\cong$ denotes an appropriate equivalence relation on processes.

This can be understood as follows. Take an arbitrary pair of traces of the system S. If these are equivalent in terms of the Low level events (but may differ in terms of High events) then it must be the case that, from the Low point of view, the system after executing t is indistinguishable from S after executing t′. Thus, Low's interactions with the system are independent of any High activity. As a consequence, Low cannot from his observations of the system infer anything about the High activity.

Note that the choice of system abstraction A_L and process equivalence $\cong$ are crucial here. Indeed, there is still no consensus in the community as to which form of process equivalence is appropriate. Furthermore, the question of how to characterise the equivalence of processes is a deep and controversial one in theoretical computer science and is closely related to the question of what exactly we mean by a "process". Many forms of equivalence have been proposed, some denotational in style, some operational, with no agreement as to which is "correct". It is perhaps unsurprising then that the security community has yet to agree the "correct" definition of confidentiality.

6.3.3 Relationship between the models

Both of these models have a role to play in the development of a secure system. Bell LaPadula is perhaps best thought of as a framework for expressing security policy, or more precisely, access control policy. Non-interference by contrast is really a framework to characterise the absence of information flow and so give a precise, formal definition of the property of confidentiality.

Characterising the absence of information flow across an interface is a key element in the analysis of a secure system. However, as we have noted, information flow is not an enforceable property. That is to say that we cannot take a system and enforce an information flow property on it by placing it in parallel with an execution monitor. This is a more formal way of stating the old dictum: security cannot be bolted on as an after-thought.

Nonetheless, the execution monitor concept is a very appealing one: from a modelling stand-point it gives rise to readily understandable and tractable models. From an implementation point of view, an EM is readily given a physical embodiment in some form of authorisation kernel that intercepts all access requests and blocks those that would violate the policy. We noted in Section 3.1.3.5 that by composing a suitable EM in parallel with an arbitrary system we can ensure that the resulting, composed system upholds the required access control policy. This means that the details of the behaviour of the (components of) the system are irrelevant to the analysis and we need only verify the EM, or its kernel embodiment. This has the great advantage that such a kernel can typically be a rather small, possibly distributed, component of the system and so it is more feasible to perform a rigorous verification on it. We still have to worry about whether the kernel is by-passable but this can be dealt with separately.

To implement an information flow policy, we could proceed by proposing a design and then perform a thorough analysis to show that the space of all possible executions has the required characteristics. In practice, this is infeasible. The kinds of system that we are interested in are typically far too large for such an analysis to be tractable. Besides, we are assuming that we

can construct a model for the analysis that is faithful to all the security relevant details. For confidentiality, where every possible observable is a potential source of information to the adversary, this is really unthinkable. To guarantee that our model is faithful, it would have to exactly replicate the real artefact and thus would be of no use as a mathematical model. A further difficulty is that, even if we supposed that our model was accurate, the fielded device would inevitably evolve. It would, for example, be upgraded and re-configured, parts would malfunction in unpredictable ways, and so forth.

Access controls, however, are enforceable and quite easy to understand, implement and verify. In practice therefore, most secure systems have been designed by postulating an architecture and using access control mechanisms. A post hoc analysis is then performed to demonstrate that the implementation upholds the information flow requirements. Such demonstrations are based on a number of assumptions and approximations. For example, the architecture is assumed known, i.e., that the channels, interfaces and protocols are all correctly identified and modelled. Recognition that such arguments are highly error prone lead to the development of covert channels analysis, as a systematic way of identifying all channels in an architecture [Lampson 1973]. Often it is simple assumed that any covert channels are of sufficiently low channel capacity to be ignorable.

We see then that there is a tension between these two styles of model: information flow is the ideal formulation of secrecy but is difficult, arguably impossible, to implement and verify. Access control, by contrast, can only give an approximation to secrecy but is comparatively easy to understand, implement and verify. In practice then, the design and analysis of secure systems is, or rather should be, performed using a mix of the two styles.

In one sense, we can think of information flow (non-interference) models as providing an underpinning to access control models like Bell-LaPadula. The Bell-LaPadula model depends heavily on informal intuitions such as the meaning of terms like *read* and *write*. In particular, it is assumed that these constitute one-way information flows. In principle at least, we could (and should) perform an analysis of the implementation of the *read* and *write* operations and establish that they really are one-way flows. In reality, they almost certainly won't be: there will typically be some back-flow to throttle data rates, for example.

6.4 Overview of specification and verification in the MAFTIA context

So far we have surveyed how the basic concepts of MAFTIA can be formalised, and in particular where new work was done in MAFTIA with respect to such formalisations. Now we give a brief general overview of where verification is useful in a MAFTIA context.

6.4.1 By security methods

Ten classes of security methods were identified in Section 3.4; here we discuss how each relates to verification and assessment work within MAFTIA:

1. *Attack prevention* (in the human sense) does not relate naturally to the type of rigorous formalism undertaken in WP6. Although one could well envisage the use of economic, social and psychological models to aid in the prevention of human attacks, we consider these outside the scope of our work.
2. *Attack prevention* (in the technical sense) could be supported by rigorous models that provide precise foundations for risk analysis. This could be achieved by specifying and verifying not only the really intended service, but also "services" with a certain maximum gain for an adversary that the system keeps up under certain weaker failure assumptions, and by trying to evaluate the investment needed for an adversary to achieve these failures.

3. *Vulnerability prevention* is the process by which one attempts to avoid introducing vulnerabilities during design. Formal, or even semi-formal, specification can help a great deal in this process since such specifications are less ambiguous than natural language.
4. *Intrusion tolerance* is the classical area where systems are proven in all the sub-fields of dependability, e.g., fault-tolerant protocols and cryptography. In particular, fault masking is accessible to verification, and also classical fault diagnosis techniques. When one can define a particular class of intrusions, then the detection of that class might be verifiable. However, since it not feasible to specify all possible types of attacks that the system may be subjected to, such verification cannot be generalised. This does mean though that we cannot show that particular systems cannot be compromised under particular assumptions.
5. As for *attack prevention*, there is no specific verification and assessment work that relates to *attack removal* (in the human sense). Instead, assuming that *attack forecasting* (in the human sense) has been undertaken, then our formalisms should account for that forecasting implicitly in their trust models.
6. *Attack removal* (in the technical sense) can benefit largely from model checking. Once debugging has identified or confirmed an attack, then the original model can be amended (usually quite quickly) to incorporate a prospective `fix', then run again to confirm that it indeed removes the attack.
7. *Vulnerability removal* is the process of verification and assessment as such. The model-checking verification technique (where applicable) is particularly useful since it provides debugging information identifying faults. Model checking can be utilised throughout the development cycle of MAFTIA to provide a means of fault removal from an early stage, in addition to a positive verification of the final product. A good example of this can be found in the modelling of two of the TTCB security services (as detailed in [Adelsbach & Creese 2003]).
8. As for *attack prevention*, *attack forecasting* (in the human sense) would seem to be more accessible to economic, social and psycological models.
9. *Attack forecasting* (in the technical sense) and *vulnerability forecasting* can be undertaken to some degree by statistical models or fault tree analysis [Welch et al. 2003]. One could also envisage the use of probabilistic model checking in this area, although this has not been considered in MAFTIA. In addition if one has in mind specific potential attacks, then it is possible to inject faults into existing models of the system to see whether it is indeed vulnerable to that attack.

6.4.2 By system life-cycle

Dependability assessment of an actual system can be structured by the system life cycle, as in the assurance part of [DIS 15408-1-3], and all phases have to be considered. Verification, however, concentrates on the design phase. That, in its turn, may go through several phases of successively detailed designs, which all need assessment.

Specific verification work in MAFTIA concentrates on those design phases where specific dependability measures are implemented, typically detailed design. High-level design (e.g., an architecture as such) is typically not detailed enough for formalisation, whereas standard hardware or software verification techniques for atomic components can be used to verify the implementation of the detailed design.

6.4.3 By architectural component

An important pre-condition for all component-wise verification is that one can indeed compose and prove systems using only the specifications of its sub-components, while preserving the proven properties, when substituting the sub-component’s specifications by according implementations, which provably fulfil the specifications. Our formalisation of the

MAFTIA basic concepts includes the proof of such a composition theorem (see Section 2.8 of [Adelsbach & Pfitzmann 2001], and Section 2.5 of [Adelsbach & Steiner 2002]). Concentrating now on fault tolerance in the detailed design, the different components of a system structured according to the MAFTIA frameworks and architecture need different verification techniques.

In Section 1.3 of [Creese & Simmonds 2002], the reader will find a discussion on composition (or compose-ability) in the context of automated proofs using CSP. Briefly, in the context of particular verifications, the components of a modular system can be considered atomic, "black-box" nodes connected by channels regulated by different CSP processes depending on the synchrony model. It is discussed how different mechanisms can help us reason about such modular systems according to node topology. Often it is the case that the node topologies correspond to a partitioning of a hybrid system into sub-systems according to, for example, failure assumptions, or underlying synchrony assumptions. Splitting a distributed system up like this not only makes for more tractable models, it can also help us to reason about global dependability properties. An example of this is can be found in the CSP modelling of two TTCB security services (see Chapter 4 of [Adelsbach & Creese 2003]).

6.4.4 By degree of formality

One can distinguish "rigorous" definitions and proofs in the sense of mathematics (where one can mix natural language and formulas quite freely), and "formal" ones in the sense of being restricted to a specific language with specific transformation rules.[22] The benefits that come with the restrictions of a formal system are that it enables tool-support, (at least syntax checks, and at best automatic proofs). We mention the use of less "formal" vulnerability assessments using *fault tree* analysis [Welch et al. 2003].

The definitions of basic concepts in MAFTIA are all only rigorous, because we were not aware of a tool that could have supported the probabilities, polynomial-time restrictions etc. However, for non-cryptographic protocols, or given suitable abstractions of cryptography proven in a rigorous way, it might be possible to use standard tools. Hence two main issues of the verification work in MAFTIA were to work towards such abstractions of cryptography, and to extend the usage of one such standard tool towards larger systems, as we now describe.

6.5 Novel verification work within MAFTIA

The novel verification work performed within MAFTIA has pursued two strands of research: the formalisation of basic concepts and new protocols, and the extension of verification techniques.

6.5.1 Abstractions from cryptography

The goal of this work is to join definition and proof techniques from cryptography with those of a wider dependability community. A first step towards this is implicit in the general formalisation of basic concepts in a model that allows cryptographic components to be included.

Our system models allow us to split reactive systems into two layers: The lower layer is a cryptographic system whose security can be rigorously proven using standard cryptographic arguments. To the upper layer it provides an abstract (and typically deterministic) service that hides all cryptographic details. Relative to this abstract service one can verify the upper layer using existing formal methods. Since our models allow secure composition, one can conclude

[22] In the previous sections, we did not make this distinction, e.g., we said "formalize" where a "rigorous" verbal definition would have sufficed.

that the overall system is secure if the formally verified upper layer is put on top of a cryptographically verified lower layer.

The main second step is to define actual abstract specifications of important cryptographic components. Abstract means in particular that these specifications should no longer be probabilistic (unless the service itself is probabilistic, e.g., for a coin flipping protocol).

We have defined abstract specifications for two initial examples: secure point-to-point channels, in both the synchronous and the asynchronous timing model, and certified mail. In the Trusted Host for secure group key agreement, we provided a normal I/O automata that could be translated into the formal language of CSP (see [Adelsbach & Creese 2003], Chapters 2 and 3); we see no theoretical reasons why that particular example could not be transcribed into a variety of formalisms. A first example, which built upon our abstract specification of a secure point-to-point channel, implemented secure ordered channels on top of our specification and can be found in [Backes et al. 2002]. The mechanism that performed the book keeping of messages was proven secure with the theorem-prover PVS, and the security of the overall system follows by the composition theorem.

The example of secure point-to-point channels also led to certain methodologies, e.g., for including *tolerable imperfections* into a specification. These are specific services to the adversary that are necessary if one wants the specification to be implementable by efficient real systems. For example, such systems allow traffic analysis (because one does not typically want to spend bandwidth to transfer dummy traffic continually), and thus an adversary can gain some information that honest users would not gain. We hope that the development of such proof techniques will lead to faster verifications in general.

6.5.2 Model-checking large protocols

Model-checking tools can only be used directly to reason about finite state systems, and usually only those of particularly small, unrealistic, sizes. The model-checking of core MAFTIA concepts and particular services has so far required a large amount of very careful modelling, rather than particularly novel techniques for overcoming size limitations. Verifications have been obtained for the asynchronous and synchronous contract-signing protocols, and selected TTCB services.

Data-Independence [Lazic 1999, Lazic & Roscoe 1999] allows us to handle systems parameterised by types, where we might want to establish correctness independently of the size of the types. In MAFTIA there were plenty of possible case studies for data independence theory (unsurprisingly, as MAFTIA promotes redundancy through replication). An example was the IBM probabilistic ABBA protocol. For that protocol, we succeeded in expressing the threshold voting mechanism that is at the heart of ABBA (and several other related MAFTIA protocols) in a data independent form. That result was reported on in [Creese & Simmonds 2002].

6.5.3 Synchrony models and availability

As already mentioned in 6.2, selected formalisms of MAFTIA concepts and dependability concepts were expressed in CSP. Of particular interest to us were the three synchrony models and the availability properties. The synchrony modelling was exemplified in the modelling of the contract signing protocols (synchronous and asynchronous) and selected TTCB services (hybrid synchrony). These are reported on in [Adelsbach & Pfitzmann 2001], [Creese & Simmonds 2002] and [Adelsbach & Creese 2003] respectively.

For the modelling of the asynchronous contract signing protocol, we had to develop a methodology for the verification of availability properties for protocols running over asynchronous networks with a notion of eventual delivery.

The modelling of the TTCB services provided us with a good case study into how to model services running over hybrid networks. We did this by essentially decomposing the system into subsystems according to the underlying synchrony assumptions. The global dependability properties were verified by "composing" local properties of the subsystems - these being verified using FDR. During the verification of the TTCB protocols a productive dialogue ensued between the protocol author and verifier to resolve ambiguities, and, in the case of the Local Authentication Service, the verifier was able to suggest minor amendments to that service that would result in a more fault tolerant service. For the relatively straightforward TTCB Local Authentication Service, we could argue that it is data independent in the sense that the results extrapolate to an arbitrary number of local TTCBs and entities.

6.5.4 Lessons Learnt

During the course of the MAFTIA CSP modelling, the verifiers learnt many lessons on good modelling practices and the need for good specification of protocols (here "specification" is used in the sense of original protocol design). Of particular importance is the need for trace-ability in the sense of correlation between original specification and final model, also dialogue between the verifiers and protocol authors to resolve ambiguities, for example. These lessons were put into practice as the project evolved, and they may be documented in a formal report.

6.5.5 Linking the Cryptographic and CSP worlds

One, important, aim of MAFTIA WP6 was to bridge the gap between the cryptographic and automated (or tool-assisted) worlds. Here "cryptographic" pertains to the rigorous secure reactive systems theory [Pfitzmann & Waidner 2001], and "automated" pertains to model checking, specifically using CSP and FDR.

The rigorous secure reactive systems theory is a well-defined mathematical framework for faithfully modelling cryptographic systems. In that theory, both cryptographic systems are transcribed as *structures* - basically probabilistic state-transition machines with *specified ports* (user-interface channels). Any structure may be augmented to form a *configuration*, by specifying a set of users, H, and an adversary machine, A. Any configuration is *run-able* in the sense that it yields a probability space of traces in terms of the inputs/outputs over its ports, and we may refer to the "view of a system" with respect to the probability space of traces over the specified ports.

A *Trusted Host* is a structure, in the above sense, representing the "best possible service" - a specification of a service suffering tolerable (and possibly unavoidable) imperfections. In the "real world", however, any particular implementation of that service will invariably suffer from more malevolent imperfections - as it will depend, for example, on imperfect cryptographic primitives, or "lossy" communications mediums.

The question then arises as to how to measure the security of these "real world" systems against the "best possible service", when the former, if reliant on imperfect primitives, can never quite match the latter. In the world of automated verification that is usually a non-question, because the types of "imperfections" that we are referring to here are usually abstracted away by Dolev-Yao type assumptions [Dolev & Yao 1983]. In the rigorous secure reactive systems theory, however, the semantics of the cryptographic primitives is faithfully preserved, and there is a formal mathematical notion of *simulatability* – a bisimilar relationship that defines what it means for a real world system to be "as least as secure as" the best possible service.

The notion of simulatability has parallels with the notion of *refinement* in the CSP calculus. So it was natural to ask whether it would be feasible to use automated checkers, such as FDR, to prove simulatability relationships. That would be good, because proving simulatability relationships by hand requires a great deal of human effort, which is, inevitably, open to error.

However, that goal proved very ambitious. The problem is not specifically one of theory - there is no reason why configurations in the rigorous secure reactive systems theory cannot naively be transcribed in CSP or similar calculi. Rather, the problem is more one of practicalities - in that the state-space of the resulting models would invariably be intractably large for today's automatic checkers, and it is our opinion that that will remain the case for the foreseeable future.

Nevertheless, as a kind of "feasibility study", we modelled in CSP the Ideal System for Group Key Establishment that is described in Chapter 2 of [Adelsbach & Creese 2003]. Our aim was to demonstrate the viability of writing tractable models of the simpler state machines that we see in the rigorous secure reactive systems theory, and to verify meaningful properties of those machines that would otherwise have to be done by hand. As such, this work may be considered complementary to that of Backes, Jacobi and Pfitzmann [Backes et al. 2002], in which the PVS theorem-prover was used to verify certain non-cryptographic "book-keeping" mechanisms of a scheme for secure message transmission. Our modelling of the Ideal System for Group Key Establishment is reported on in detail in Chapter 3 of [Adelsbach & Creese 2003].

Chapter 7 Conclusion

This deliverable contains four main contributions:

- A discussion of the nature of security policies, rules and goals, in the context of security failures and faults, an analysis of attacks, vulnerabilities and intrusions in terms of the basic dependability concepts of fault, error and failure, and the identification of ten security methods for dealing with attacks, vulnerabilities and intrusions
- A discussion of the relationship between intrusion detection and intrusion tolerance, and the development of an integrated intrusion detection/tolerance framework for building intrusion tolerant systems
- An introduction to the MAFTIA architecture and a discussion of the underlying models and fault assumptions upon which it is based, including a description of the various strategies that are being used to build intrusion tolerant components. Three such strategies are identified, namely "fail-uncontrolled", "fail-controlled with distributed trusted components", and "fail-controlled with local trusted components".
- A brief overview of some of the issues concerning the formalisation of the MAFTIA conceptual model, a discussion of approaches to formalising security policies, and an introduction to the methods of validation and assessment that are being used as fault removal techniques to ensure the security of the protocols that are used to implement the MAFTIA architecture

Other deliverables provide more detail about MAFTIA's work on intrusion-tolerant middleware [Cachin 2001b, Neves & Veríssimo 2001], intrusion-detection systems [Dacier 2002], trusted third parties [Cachin 2002], authorisation services [Abghour et al. 2001, Abghour et al. 2002], and the work on verification and assessment of secure systems [Adelsbach & Creese 2003]. The role of this deliverable is to describe the basic concepts of dependability and intrusion tolerance that underpin all of the MAFTIA work. These concepts and architectural principles reflect the experience gained from prototyping and validating selected components of the overall MAFTIA architecture.

Glossary

This glossary is provided as an aid to reading this document. It should not be considered independently of the body of the document.

abuse of privilege – see *(privilege, abuse of ~)*.

access control – the prevention of use of a resource by unidentified and/or unauthorised entities in any other than an authorised manner [ECMA TR/46]; the determination as to whether a requested access to an information item is to be granted or denied; see also, *authorisation*.

accidental – not deliberate

accountability – availability and integrity of some meta-information related to an operation (e.g., identity of the user realising the operation, time of the operation, etc.).

alarm (intrusion-detection ~) – a report of an error that may lead to or has led to a security failure, optionally including diagnostic information about the cause of the error.

activity – *event* or a sequence of events within a given context.

anonymisation – process that gives confidence in *anonymity*.

anonymity – *confidentiality* of the identity of a person, e.g., who has realised an operation, or has not realised an operation.

attack – (general sense) a malicious interaction *fault,* through which an attacker aims to deliberately violate one or more security properties; an *intrusion* attempt; (human sense) a malicious human interaction *fault* whereby an attacker aims to deliberately violate one or more security properties; (technical sense) a malicious technical interaction *fault* aiming to exploit a *vulnerability* as a step towards achieving the final aim of the attacker

attack agent – malicious logic carrying an *attack* on behalf of an *attacker*

attacker – malicious person or organization at the origin of *attacks*

auditability – availability and integrity of some meta-information related to all operations.

authentic – of undisputed origin, genuine [OMED 1992].

authentication – process which gives confidence in *authenticity*.

authenticity – integrity of some information and meta-information; integrity of the meta-information representing the link between some information and its origin (e.g., the meta-information relating the claimed identity of a subject to the real identity of the subject).

authorisation – the granting of access to a security object [ECMA TR/46]; the determination as to whether a requested operation is to be granted or denied, according to the security policy; see also, *access control*.

availability – *dependability* with respect to the readiness for correct service; measure of *correct service* delivery with respect to the alternation between *correct service* and *incorrect service* [Laprie 1992].

component (system ~) – another system, which is part of the considered system [Laprie 1992].

confidentiality – dependability with respect to the non-occurrence of unauthorised information disclosure.

correct service – see *service (correct ~).*

coverage – measure of the representativity of the situations to which a *system* is submitted during its validation compared to the actual situations it will be confronted with during its operational life [Laprie 1992].

dependability – ability of a system to deliver a service that can justifiably be trusted.

dependence – the state of being dependent on other support [OMED 1992]; reliance, trust, confidence [OMED 1992].

dependent – depending, conditional or subordinate [OMED 1992].

error – part of the state of a *system* that may lead to subsequent *failure*; manifestation of a *fault* in a *system* [Laprie 1992].

event – a thing that happens or takes place [OMED 1992]; a change in *state.*

failure – event occurring when the delivered *service* deviates from *correct service*; see also: *security failure*.

failure model – a *fault model* defined in terms of the *failures* of the *components* of a *system*.

failure (security~) – violation of a security goal of the intended *security policy*.

false negative – the *event* corresponding to the incorrect decision not to rate an activity as being erroneous; also called a "miss".

false positive – the *event* corresponding to the incorrect decision to rate an activity as being erroneous; also called a "false alarm".

fault – adjudged or hypothesised cause of an *error*; *error* cause which is intended to be avoided or tolerated [Laprie 1992]; consequence for a *system* of the *failure* of another *system* which has interacted or is interacting with the considered *system* [Laprie 1992].

fault forecasting – see *forecasting (fault ~).*

fault model – set of assumptions about the *faults* that are taken into account during *fault prevention*, *tolerance*, *removal* or *forecasting.*

fault prevention – see *prevention (fault ~).*

fault removal – see *removal (fault ~).*

fault tolerance – see *tolerance (fault ~)*.

forecasting (fault ~) – how to estimate the present number, the future incidence, and the likely consequences of *fault.s*

forecasting (attack ~) – how to estimate the present number, the future incidence, and the likely consequences of *attacks.*

forecasting (vulnerabality ~) – how to estimate the present number, the future incidence, and the likely consequences of *vulnerabilities.*

identity – representation of a person in a *system*.

incorrect service – see *service (incorrect ~).*

insider – a human *user* authorised to perform some of a set of specified operations on a set of specified objects, i.e., a user whose (current) *privilege* intersects the considered domain of object-operation pairs.

insider intrusion – see *intrusion (insider ~).*

intruder – malicious person or organization at the origin of *intrusions*, i.e., an *attacker* that has successfully exploited a *vulnerability*

integrity – absence of improper state alterations.

intrusion – a *malicious*, externally-induced *fault* resulting from an *attack* that has been successful in exploiting a *vulnerability.*

intrusion (insider ~) – an *abuse of privilege*.

intrusion (outsider ~) – a *theft of privilege*.

intrusion detection: concerns the set of practices and mechanisms used towards detecting *errors* that may lead to *security failure*, and diagnosing *intrusions* and *attacks*.

intrusion-detection system: an implementation of the practices and mechanisms of *intrusion detection*.

logic bomb – *malicious logic* that remains dormant in the host system till a certain time or an event occurs, or certain conditions are met, and then deletes files, slows down or crashes the host system, etc.

malicious – intending or intended to do harm [OMED 1992].

malicious logic – an internal, deliberately malicious fault; malicious logic may by a *logic bomb*, a *zombie*, a *Trojan horse*, a *trapdoor*, a *virus*, a *worm*, an illegal *sniffer*, etc.

misfeasance – the illegal or improper performance of an action in itself lawful [LMED 1976]; an *intrusion* through the abuse of *privilege*.

object – information container.

outsider – a human *user* not authorised to perform any of a set of specified operations on a set of specified objects, i.e., a user whose (current) *privilege* does not intersect the considered domain of object-operation pairs.

outsider intrusion – see *(intrusion, outsider ~).*

prevention (fault ~) – how to prevent the occurrence or introduction of *faults*

prevention (attack ~) – how to prevent the occurrence of (human or technical) *attacks*.

prevention (intrusion ~) – how to prevent the occurrence of *intrusions* (through *attack prevention*, *vulnerability prevention* and *vulnerability removal*).

prevention (vulnerability ~) – how to prevent the occurrence or introduction of *vulnerabilities*.

privacy – *confidentiality* of personal information.

privilege – set of *rights* of a *subject.*

privilege (abuse of ~) – a *misfeasance*, i.e., an improper use of authorised operations.

privilege (theft of ~) – an unauthorised increase in privilege, i.e., a change in the *privilege* of a user that is not permitted by the system's security policy.

removal (fault ~) – how to reduce the number or severity of *faults*.

removal (attack ~) – how to reduce the number or severity of (human or technical) *attacks*

removal (vulnerability ~) – how to reduce the number or severity of *vulnerabilities*

responsibility – the state of being responsible [OMED 1992].

responsible – obliged to account; being the cause of; accountable for.

rights – a subject has a given right on a specified *object* if and only if he is authorised to perform a specified operation on that object; elements of a subject's *privilege*.

security – *dependability* with respect to the prevention of unauthorised access and/or handling of information [Laprie 1992]; the combination of *confidentiality*, *integrity* and *availability*.

security failure – see *failure (security~)*.

security policy – description of 1) the security properties which are to be fulfilled by a computing system; 2) the rules according to which the system security state can evolve.

service – *system* behaviour as perceived by a *system user*.

service (correct ~) – *service* that implements the *system function*.

service (incorrect ~) – *service* that does not implement the *system function*.

sniffer – a program that monitors network traffic.

state (system ~) – a condition of being with respect to a set of circumstances [Laprie 1992].

subject – active entity in a computer *system* — a process is a subject, a human *user* is also a subject.

system – entity having interacted, interacting or able to interact with other entities [Laprie 1992]; set of components bound together in order to interact [Laprie 1992].

system function – that for which the *system* is intended.

system user – see *user (system ~)*.

theft of privilege – see *(privilege, theft of ~)*.

tolerance (fault ~) – how to provide correct service in the presence of *faults*

tolerance (intrusion ~) – how to provide correct service in the presence of *intrusions*

trapdoor – *malicious logic* that provides a means of circumventing access control mechanisms.

Trojan horse – *malicious logic* performing an illegitimate action while giving the impression of being legitimate; the illegitimate action can be the disclosure or modification of information (attack against confidentiality or integrity) or a *logic bomb*.

true negative – the *event* corresponding to the correct decision not to rate an activity as being erroneous.

true positive – the *event* corresponding to the correct decision to rate an activity as being erroneous.

true negative – the *event* corresponding to the correct decision of an *intrusion-detection system* to rate an observed activity as not being malicious.

true positive – the *event* corresponding to an alarm correctly generated by an *intrusion-detection system*.

trust – reliance on the truth of a statement etc. without examination [OMED 1992].

trusted – adjective to describe a statement etc. on which *trust* has been placed.

user (system ~) – another *system* (physical, human) interacting with the considered *system*.

virus – *malicious logic* that replicates itself and joins another program (system or application) when it is executed, thereby turning into a *Trojan horse*; a virus can carry a *logic bomb*.

vulnerability – a fault created during development of the system, or during operation, that could be exploited to create an *intrusion*.

worm – *malicious logic* that replicates itself and propagates without the users being aware of it; a worm can also carry a *logic bomb*.

zombie – a *logic bomb* that can be triggered by an attacker in order to mount a coordinated attack.

References

[Abghour et al. 2001] N. Abghour, Y. Deswarte, V. Nicomette and D. Powell, *Specification of Authorisation Services*, MAFTIA Project, Deliverable D27, January 2001.

[Abghour et al. 2002] N. Abghour, Y. Deswarte, V. Nicomette and D. Powell, *Design of the Local Reference Monitor*, MAFTIA Project, Deliverable D6, April 2002.

[Abrams et al. 1995] M. Abrams, S. Jajodia and H. Podell, *Information Security,* IEEE CS Press, 1995.

[Adelsbach & Pfitzmann 2001] A. Adelsbach and B. Pfitzmann (Eds.), *Formal Model of Basic Concepts,* MAFTIA Project, Deliverable D4, 2001.

[Adelsbach & Steiner 2002] A. Adelsbach and M. Steiner (Eds.), *Cryptographic Semantics for Algebraic Model,* MAFTIA Project, Deliverable D8, 2002.

[Adelsbach & Creese 2003] A. Adelsbach and S. Creese (Eds.), *Final Report on Verification and Assessment,* MAFTIA Project, Deliverable D22, 2003.

[Alessandri 2001] D. Alessandri (Ed.), *Towards a Taxonomy of Intrusion Detection Systems and Attacks,* MAFTIA Project, Deliverable D3, 2001.

[Alpern & Schneider 1985] B. Alpern and F. B. Schneider, "Defining Liveness", *Information Processing Letters*, 21, pp.181-85, 1985.

[Anderson 2001] R. J. Anderson, *Security Engineering: a Guide to Building Dependable Distributed Systems,* Wiley Computer Publishing, 2001.

[Anderson & Lee 1981] T. A. Anderson and P. A. Lee, *Fault Tolerance — Principles and Practice,* Prentice-Hall, 1981 (see also: P.A. Lee, T. Anderson, *Fault Tolerance - Principles and Practice*, Dependable Computing and Fault-Tolerant Systems, vol. 3, Springer-Verlag, Vienna, 1990).

[Asokan et al. 2000] N. Asokan, B. Baum-Waidner, T. Pedersen, B. Pfitzmann, M. Schunter, M. Steiner and M. Waidner, "Architecture", in *SEMPER - Secure Electronic Marketplace for Europe* LNCS 1854, pp.45-63, Springer-Verlag, Berlin, 2000.

[Avizienis 1967] A. Avizienis, "Design of Fault-Tolerant Computers", in *Fall Joint Computer Conference,* AFIPS Conf. Proc., 31, pp.733-43, Washington D.C.: Thompson Books, 1967.

[Avizienis & Chen 1977] A. Avizienis and L. Chen, "On the implementation of N-version programming for software fault tolerance during execution", in *First IEEE-CS International Computer Software and Applications Conference (COMPSAC 77),* (Chicago, IL, USA), pp.149-55, IEEE CS Press, 1977.

[Avizienis et al. 2001] A. Avizienis, J.-C. Laprie and B. Randell, *Fundamental Concepts of Dependability*, LAAS-CNRS, Research Report 01145 (Revision 1: December 2002), August 2001 (UCLA CSD Report no. 010028; Newcastle University Report no. CS-TR-739).

[Babaõglu 1987] O. Babaõglu, "On the Reliability of Consensus-Based Fault-Tolerant Distributed Computing Systems", *ACM Transactions on Computer Systems*, 5 (3), pp.394-416, 1987.

[Backes 2002] M. Backes, *Cryptographically Sound Analysis of Security Protocols,* PhD, Universität des Saarlandes, 2002.

[Backes et al. 2002] M. Backes, C. Jacobi and B. Pfitzmann, "Deriving Cryptographically Sound Implementations Using Composition and Formally Verified Bisimulation",

in *Formal Methods Europe (FME'02),* (Kopenhagen, Denmark), LNCS, pp.310-29, Springer-Verlag, 2002.

[Baker et al. 1995] W. E. Baker, R. W. Horst, D. P. Sonnier and W. J. Watson, "A Flexible ServerNet-based Fault-Tolerant Architecture", in *25th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-25),* (Pasadena, CA, USA), pp.2-11, IEEE CS Press, 1995.

[Bell & LaPadula 1974] D. E. Bell and L. J. LaPadula, *Secure Computer Systems: Unified Exposition and Multics Interpretation*, The MITRE Corporation, Report MTR-2997 (AD/A-020-445), April 1974.

[Blain & Deswarte 1990] L. Blain and Y. Deswarte, "An Intrusion-Tolerant Security Server for an Open Distributed System", in *European Symp. on Research in Computer Security,* (Toulouse, France), pp.97-104, AFCET, 1990.

[Bossen & Hsiao 1980] D. C. Bossen and M. Y. Hsiao, "A System Solution to the Memory Soft Error Problem", *IBM Journal of Research and Development*, 24 (3), pp.390-97, May 1980.

[Bouricius et al. 1969] W. G. Bouricius, W. C. Carter and P. R. Schneider, "Reliability Modeling Techniques for Self-Repairing Computer Systems", in *24th National Conference,* pp.295-309, ACM, 1969.

[Bowen et al. 1997] N. S. Bowen, J. Antognini, R. D. Regan and N. C. Matsakis, "Availability in Parallel Systems: Automatic Process Restart", *IBM Systems Journal*, 36 (2), pp.284-300, 1997.

[Bracha & Toueg 1985] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols", *Journal of the ACM*, 32 (4), pp.824-40, October 1985.

[Brière & Traverse 1993] D. Brière and P. Traverse, "AIRBUS A320/A330/A340 Electrical Flight Controls - A Family of Fault-Tolerant Systems", in *23rd IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-23),* (Toulouse, France), pp.616-23, IEEE CS Press, 1993.

[Brown Associates 1998] *Competitive Analysis of Reliability, Availability, Serviceability and Cluster Features and Functions*, D.H. Brown Associates, Inc., Report.

[Cachin 2001a] C. Cachin, "Distributing Trust on the Internet", in *Int. Conf. on Dependable Systems and Networks (DSN-2001),* (Goteborg, Sweden), IEEE CS Press, 2001a.

[Cachin 2001b] C. Cachin (Ed.), *Specification of Dependable Trusted Third Parties,* MAFTIA Project, Deliverable 26, 2001b.

[Cachin 2002] C. Cachin (Ed.), *Full Design of Dependable Third Party Services,* MAFTIA Project, Deliverable D5, 2002.

[CEN 13608-1] *Health Informatics - Security for Healthcare Communication - Part 1: Concepts and Terminology*, CEN, European Prestandard.

[Chandra & Toueg 1996] R. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems", *Journal of the ACM*, 43 (2), pp.225-67, March 1996.

[Clark & Wilson 1987] D. D. Clark and D. R. Wilson, "A Comparison of Commercial and Military Computer Security Policies", in *Symp. on Security and Privacy,* (Oakland, CA, USA), pp.184-94, IEEE CS Press, 1987.

[Cohen 1977] E. Cohen, "Information Transmission in Computational Systems", in *Sixth ACM Symp. On Operating Systems Principles,* pp.133-39, 1977.

[Cramp et al. 1992] R. Cramp, M. A. Vouk and W. Jones, "On Operational Availability of a Large Software-based Telecommunications System", in *3rd Int. Symp. on Software*

Reliability Engineering (ISSRE'92), (Research Triangle Park, NC, USA), pp.358-66, IEEE CS Press, 1992.

[Creese & Simmonds 2002] S. Creese and W. Simmonds (Eds.), *Specification and Verification of Selected Intrusion Tolerance Properties using CSP and FDR,* MAFTIA Project, Deliverable D7, 2002.

[Dacier 2002] M. Dacier (Ed.), *Design of an Intrusion-Tolerant Intrusion Detection System,* MAFTIA Project, Deliverable D10, 2002.

[Debar et al. 1999] H. Debar, M. Dacier and A. Wespi, "Towards a Taxonomy of Intrusion-Detection Systems", *Computer Networks*, 31 (8), pp.805-22, 1999.

[Deswarte et al. 1991] Y. Deswarte, L. Blain and J.-C. Fabre, "Intrusion Tolerance in Distributed Systems", in *Symp. on Research in Security and Privacy,* (Oakland, CA, USA), pp.110-21, IEEE CS Press, 1991.

[Deswarte et al. 2001] Y. Deswarte, N. Abghour, V. Nicomette and D. Powell, "An Internet Authorization Scheme using Smart Card-Based Security Kernels", in *e-Smart 2001,* (I.Attali and T.Jensen, Eds.), (Cannes, France), Lecture Notes in Computer Science, 2140, pp.71-82, 2001.

[Deswarte et al. 2002] Y. Deswarte, N. Abghour, V. Nicomette and D. Powell, "An Intrusion-Tolerant Authorization Scheme for Internet Applications", in *International Conference on Dependable Systems and Networks (DSN'2002),* (Washington, D.C. (USA)), pp.C.1.1-C.1.6, IEEE Computer Society Press, 2002.

[DIS 15408-1-3] *Common Criteria for Information Technology Security Evaluation*, Common Criteria Project Sponsoring Organisations, May 1998, adopted by ISO/IEC as Draft International Standard DIS 15408 1-3.

[Dobson & Randell 1986] J. E. Dobson and B. Randell, "Building Reliable Secure Systems out of Unreliable Insecure Components", in *Conf. on Security and Privacy,* (Oakland, CA, USA), pp.187-93, IEEE CS Press, 1986.

[Dolev & Yao 1983] D. Dolev and A. C. Yao, "On the Security of Public Key Protocols", *Ieee Transactions on Information Theory*, 29 (2) 1983.

[ECMA TR/46] ECMA TR/46, *Security in Open Systems — A Security Framework*, ECMA, Technical Report 46, 1988.

[Ellison et al. 1997] R. J. Ellison, D. A. Fischer, R. C. Linger, H. F. Lipson, T. Longstaff and N. R. Mead, *Survivable Network Systems: an Emerging Discipline*, Technical Report CMU/SEI-97-TR-013, November 1997 (revised May 1999).

[Feiertag 1980] R. J. Feiertag, *A Technique for Proving Specifications are Multi-level Secure*, CSL, SRI International, Technical Report CSL109, 1980.

[Ferraiolo & Kuhn 1992] D. Ferraiolo and R. Kuhn, "Role-Based Access Control", in *15th National Computer Security Conference,* pp.554-63, NIST, 1992.

[Focardi & Martinelli 1999] R. Focardi and F. Martinelli, "A uniform approach for the definition of security properties", in *World Congress on Formal Methods (FM'99),* LNCS, pp.794-813, Springer-Verlag, 1999.

[Formal Systems (Europe) Ltd] Formal Systems (Europe) Ltd, "Failures-Divergences Refinement" (accessed: 17 October 2000) http://www.formal.demon.co.uk/FDR2.html

[Fraga & Powell 1985] J. Fraga and D. Powell, "A Fault and Intrusion-Tolerant File System", in *IFIP 3rd Int. Conf. on Computer Security,* (J. B. Grimson and H.-J. Kugler, Eds.), (Dublin, Ireland), Computer Security, pp.203-18, Elsevier Science Publishers B.V. (North-Holland), 1985.

[Fray et al. 1986] J.-M. Fray, Y. Deswarte and D. Powell, "Intrusion-Tolerance using Fine-Grain Fragmentation-Scattering", in *Symp. on Security and Privacy,* (Oakland, CA, USA), pp.194-201, IEEE CS Press, 1986.

[Garber 2000] L. Garber, "Denial-of-Service Attacks Rip the Internet", *Computer*, 33 (4), pp.12-17, April 2000.

[Goguen & Meseguer 1982] J. A. Goguen and J. Meseguer, "Security Policies and Security Models", in *1982 Symp. on Security and Privacy,* (Oakland, CA), pp.11-20, IEEE CS Press, 1982.

[Gollmann 1999] D. Gollmann, *Computer Security,* John Wiley and Sons, Inc, New York, 1999.

[Gong 1992] L. Gong, "A Security Risk of Depending on Synchronized Clocks", *Operating Systems Review*, 26 (1), pp.49-53, January 1992.

[Gray III 1992] J. Gray III, "Towards a Mathematical Foundation for Information Flow Security", *Journal of Computer Security*, 1 (3,4), pp.255-94, 1992 1992.

[Gray 1978] J. Gray, "Notes on Database Operating Systems", in *Operating Systems: An Advanced Course* (R. Bayer, R. M. Graham and G. Seegmuller, Eds.), Lecture Notes in Computer Science, 60, pp.393-481, Springer-Verlag, Berlin, 1978.

[Gray 1990] J. Gray, "A Census of Tandem System Availability Between 1985 and 1990", *IEEE Transactions on Reliability*, R-39 (4), pp.409-18, 1990.

[Halme & Bauer] L. R. Halme and R. K. Bauer, "AINT Misbehaving: A Taxonomy of Anti-Intrusion Techniques" (accessed: 10 April 2000) http://www.sans.org/newlook/resources/IDFAQ/aint.htm

[Härder & Reuter 1983] T. Härder and A. Reuter, "Principles of Transaction-Oriented Database Recovery", *Computing Surveys*, 15 (4), pp.287-317, December 1983.

[Hennebert & Guiho 1993] C. Hennebert and G. Guiho, "SACEM: A Fault-Tolerant System for Train Speed Control", in *23rd IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-23),* (Toulouse, France), pp.624-28, IEEE CS Press, 1993.

[Hoare 1985] C. A. R. Hoare, *Communicating Sequential Processes,* Prentice-Hall, 1985.

[IETF] IETF, "Intrusion Detection Exchange Format" (accessed: 5 September 2001) http://www.ietf.org/html.charters/idwg-charter.html

[ISO 7498-2] *Basic Reference Model, Part 2: Security Architecture*, ISO, IS.

[ITSEC] *Information Technology Security Evaluation Criteria*, Commission of the European Communities, Office for Official Publications of the European Communities, June 1991.

[Jones 2000] A. Jones, "The Challenge of Building Survivable Information-Intensive Systems", *IEEE Computer*, 33 (8), pp.39-43, August 2000.

[Joseph & Avizienis 1988] M. K. Joseph and A. Avizienis, "A Fault Tolerance Approach to Computer Viruses", in *1988 Symp. on Security and Privacy,* (Oakland, CA, USA), pp.52-58, IEEE CS Press, 1988.

[Kantz & Koza 1995] H. Kantz and C. Koza, "The ELEKTRA Railway Signalling System: Field Experience with an Actively Replicated System with Diversity", in *25th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-25),* (Pasadena, CA, USA), pp.453-58, IEEE CS Press, 1995.

[Kent & Atkinson 1998] S. Kent and R. Atkinson, *Security Architecture for the Internet Protocol*, IETF, Technical Report Request for Comments 2401 November 1998.

[Kuhn 1997] D. R. Kuhn, "Sources of Failure in the Public Switched Telephone Network", *IEEE Computer*, 30 (4), pp.31-36, April 1997.

[Lampson 1973] B. Lampson, "A Note on the Confinement Problem", *Communications of the ACM*, 16 (10), pp.613-15, October 1973.

[Landwehr et al. 1994] C. E. Landwehr, A. R. Bull, J. P. McDermott and W. S. Choi, "A Taxonomy of Computer Program Security Flaws", *ACM Computing Surveys*, 26 (3), pp.211-54, September 1994.

[Laprie 1985] J.-C. Laprie, "Dependable Computing and Fault Tolerance: Concepts and Terminology", in *15th IEEE Int. Symp. on Fault Tolerant Computing (FTCS-15),* (Ann Arbor, MI, USA), pp.2-11, IEEE CS Press, 1985.

[Laprie 1992] J.-C. Laprie (Ed.), *Dependability: Basic Concepts and Terminology,* Dependable Computing and Fault-Tolerance, 5, 265p., Springer-Verlag, Vienna, Austria, 1992.

[Lazic 1999] R. S. Lazic, *A Semantic Study of Data Independence with Applications to Model Checking,* DPhil Thesis, Oxford University, 1999.

[Lazic & Roscoe 1999] R. S. Lazic and A. W. Roscoe, "Data Independence with Predicate Symbols", in *Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99),* (Las Vegas, Nevada, USA), Volume 1, pp.319-25, CSREA Press, 1999.

[LMED 1976] *Longman Modern English Dictionary*, Longman Group Limited1976.

[MAFTIA 2000] MAFTIA, *Reference Model and Use Cases*, MAFTIA Project, Deliverable D1, 2000.

[McLean 1994] J. McLean, "Security Models", in *Encyclopaedia of Software Engineering* Wiley Computer Publishing, 1994.

[Menezes et al. 1996] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone, *Handbook of Applied Cryptography,* CRC Press, 1996.

[Meyer & Pradhan 1987] F. Meyer and D. Pradhan, "Consensus with Dual Failure Modes", in *Digest of Papers, The 17th Int. Symp. on Fault-Tolerant Computing Systems,* (Pittsburgh, USA), pp.214-22, IEEE CS, 1987.

[Miranda et al. 2001] H. Miranda, A. Pinto and L. Rodrigues, "Appia, a flexible protocol kernel supporting multiple coordinated channels", in *Proc. 21st International conference on Distributed Computing Systems (ICDCS-21),* (Phoenix, Arizona, USA), pp.707-10, IEEE Computer Society, 2001.

[Neves & Veríssimo 2001] N. F. Neves and P. Veríssimo (Eds.), *First Specification of APIs and Protocols for the MAFTIA Middleware,* MAFTIA Project, Deliverable D24, 2001.

[Neves & Veríssimo 2002] N. F. Neves and P. Veríssimo (Eds.), *Complete Specification of APIs and Protocols for the MAFTIA Middleware,* MAFTIA Project, Deliverable D9, 2002.

[NSA 1998] NSA, 1998, "NSA Glossary of Terms Used in Security and Intrusion Detection", National Security Agency (accessed: 5 September 2001) http://www.sans.org/newlook/resources/glossary.htm

[OMED 1992] *The Oxford Modern English Dictionary*, Oxford University Press,1992 (J. Swannel, Ed.).

[Pfitzmann & Waidner 1994] B. Pfitzmann and M. Waidner, *A General Framework for Formal Notions of 'Secure' Systems*, Universität Hildesheim, Report 11/94, April 1994 (ISSN 0941-3014).

[Pfitzmann & Waidner 2001] B. Pfitzmann and M. Waidner, "A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission", in *IEEE Symp. on Security and Privacy,* (Oakland, CA, USA), pp.184-200, 2001.

[Porras et al.] P. Porras, D. Schnackenberg, S. Staniford-Chen and M. Stillman, "The Common Intrusion Detection Framework Architecture", CIDF working group (accessed: 5 September 2001) http://www.gidos.org/drafts/architecture.txt

[Powell et al. 1988] D. Powell, G. Bonn, D. Seaton, P. Veríssimo and F. Waeselynck, "The Delta-4 Approach to Dependability in Open Distributed Computing Systems", in *18th IEEE Int. Symp. on Fault-Tolerant Computing Systems (FTCS-18),* (Tokyo, Japan), pp.246-51, IEEE CS Press, 1988

[Powell & Stroud 2001] D. Powell and R. J. Stroud (Eds.), *Conceptual Model and Architecture,* MAFTIA Project, Deliverable D2, 2001.

[Rabin 1989] M. O. Rabin, "Efficient Dispersal of Information for Security, Load Balancing and Fault Tolerance", *Journal of the ACM*, 36 (2), pp.335-48, April 1989.

[Randell 1975] B. Randell, "System Structure for Software Fault Tolerance", *IEEE Transactions on Software Engineering*, SE-1 (2), pp.220-32, 1975.

[Rodrigues et al. 1996] L. Rodrigues, K. Guo, A. Sargento, R. v. Renesse, B. Glade, P. Verissimo and K. Birman, "A Transparent Light-Weight Group Service", in *Proc. 15th Int. Conf. on Fault-Tolerant Computing Systems (FTCS-15),* (Niagra-on-the-Lake, Canada), pp.130-39, IEEE CS, 1996.

[Rodrigues & Verissimo 2000] L. Rodrigues and P. Verissimo, "Topology-aware Algorithms for Large-scale Communication", in *Recent Advances in Distributed Systems* (S. Krakowiak and S. K. Shrivastava, Eds.), LNCS vol. 1752, Springer-Verlag, 2000 (Chapter 6, Part I - Distributed Algorithms).

[Roscoe 1998] A. W. Roscoe, *The Theory and Practice of Concurrency,* Prentice-Hall, 1998.

[Ruegger 1990] B. Ruegger, *Human Error in the Cockpit*, Swiss Reinsurance Company, 1990.

[Ryan 2000] P. Y. A. Ryan, "Mathematical Models of Computer Security", in *Foundations of Security Analysis and Design (FOSAD'2000),* (R. Foccardi and R. Gorrieri, Eds.), LNCS, pp.1-62, Springer-Verlag, 2000.

[Samarati & Vimercati 2000] P. Samarati and S. d. C. d. Vimercati, "Access Control: Policies, Models and Mechanisms", in *Foundations of Security Analysis and Design (FOSAD'2000),* (R. Foccardi and R. Gorrieri, Eds.), LNCS, pp.137-96, Springer-Verlag, 2000.

[Schneider 1999] F. Schneider (Ed.), *Trust in Cyberspace,* National Academy Press, 1999.

[Schneider 2000] F. Schneider, "Enforceable security policies", *ACM Transactions on Information and System Security*, 3 (1), pp.30-50, February 2000.

[Schneier 1996] B. Schneier, *Applied Cryptography,* Wiley and Sons, Inc., 1996.

[Schulzrinne et al. 1996] H. Schulzrinne, S. Casner, R. Frederick and V. Jacobson, *RTP: A Transport Protocol for Real-Time Applications*, Audio-Video Transport Working Group, Techical Report Request for Comment 1889 January 1996.

[Smith 1986] T. B. Smith, "High Performance Fault-Tolerant Real-Time Computer Architecture", in *16th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-16),* (Vienna, Austria), pp.14-19, IEEE CS Press, 1986.

[Trouessin 2000] G. Trouessin, *Towards Trustworthy Security for Healthcare Information Systems*, CESSI/CNAM, Report GT/2000.03, June 2000.

[Verissimo et al. 1997] P. Verissimo, L. Rodrigues and A. Casimiro, "Cesiumspray: a Precise and Accurate Global Time Service for Large-Scale Systems", *Journal of Real-Time Systems*, 12 (3), pp.243-94, May 1997.

[Verissimo et al. 2000] P. Verissimo, N. F. Neves and M. Correia, "The Middleware Architecture of MAFTIA", in *3rd Information Survivability Workshop,* (Boston, MA, USA), IEEE CS Press, 2000.

[Welch et al. 2003] I. S. Welch, J. P. Warne, P. Y. A. Ryan and R. J. Stroud, *Architectural Analysis of MAFTIA's Intrusion Tolerance Capabilities*, MAFTIA Project, Deliverable D99, January 2003.

[Wilson 1985] D. Wilson, "The STRATUS Computer System", in *Resilient Computing Systems* (T. Anderson, Ed.), pp.208-31, Collins, London, UK, 1985.

[Yeh 1998] Y. C. B. Yeh, "Dependability of the 777 Primary Flight Control System", in *Dependable Computing for Critical Applications (DCCA-5)* (R. K. Iyer, M. Morganti, W. K. Fuchs and V. Gligor, Eds.), Dependable Computing and Fault-Tolerant Systems, 10, pp.3-17, IEEE CS Press, 1998 (Proc. IFIP 10.4 Work. Conf. held in Urbana-Champaign, IL, USA, September 1995).

[Zurko et al. 1999] M.-E. Zurko, R. Simon and T. Sanfilipo, "A User-Centered, Modular Authorization Service Built on an RBAC Foundation", in *IEEE Symposium on Security and Privacy,* (Berkeley (CA, USA)), pp.57-71, 1999.