Project IST-1999-11583

Malicious- and Accidental-Fault Tolerance for Internet Applications

Conceptual Model and Architecture

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

MAFTIA deliverable D2

Public document

November 20, 2001

Technical Report CS-TR-749, University of Newcastle upon Tyne
Technical Report DI/FCUL TR-01-10, Universidade de Lisboa
LAAS-CNRS Report No. 01426
Research Report RZ 3377, IBM Research, Zurich Research Laboratory

List of Contributors

André Adelsbach (Universität des Saarlandes)

Dominique Alessandri (IBM ZRL)

Christian Cachin (IBM ZRL)

Sadie Creese (Qinetiq)

Marc Dacier (IBM ZRL)

Yves Deswarte (LAAS-CNRS)

Klause Kursawe (IBM ZRL)

Jean-Claude Laprie (LAAS-CNRS)

Birgit Pfitzmann (Universität des Saarlandes)

David Powell (LAAS-CNRS)

Brian Randell (University of Newcastle upon Tyne)

James Riordan (IBM ZRL)

Robert Stroud (University of Newcastle upon Tyne)

Paulo Veríssimo (Universidade de Lisboa)

Michael Waidner (IBM ZRL)

Ian Welch (University of Newcastle upon Tyne)

Andreas Wespi (IBM ZRL)

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

This deliverable builds on the work reported in MAFTIA deliverable D1 [Cachin *et al.* 2000a]. It contains a refinement of the MAFTIA conceptual model and a discussion of the MAFTIA architecture. It also introduces the work done in WP6 on verification and assessment of security properties, which is reported on in more detail in MAFTIA deliverable D4 [Adelsbach & Pfitzmann 2001].

Chapter 2 is taken largely from [Laprie *et al.* 1995] and presents core dependability concepts. Chapter 3 refines these in the context of malicious faults, and examines the distinction between intrusions, attacks, and vulnerabilities. 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.

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.

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 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 introduces the work done on verification and assessment of secure systems. This is discussed in much more detail in MAFTIA deliverable D4 [Adelsbach & Pfitzmann 2001], but again the idea is to provide an introduction or executive summary of the work. In terms of the basic dependability concepts discussed in Chapter 3, the purpose of verification and assessment is vulnerability removal.

Chapter 7 concludes the deliverable with a discussion of what has been achieved and directions for future work, and an appendix contains a glossary of the terms used in the deliverable.

Chapter 2 Core dependability concepts

The purpose of this chapter is to recall some core dependability concepts, extracted mostly literatim from [Laprie *et al.* 1995, chapter 1][1]. We then extend and refine these definitions in the context of security and intrusion-tolerance/detection. Readers familiar with the core dependability concepts may proceed directly to Chapter 3.

2.1 Basic definitions

Dependability is that property of a computer system such that *reliance can justifiably be placed on the service it delivers.* The **service** delivered by a system is its behaviour *as perceived* by its user(s); a **user** is another system (human or physical) interacting with the system considered.

According to the application(s) of the system, different facets of dependability may be highlighted. This is tantamount to stating that dependability can be viewed according to different but complementary properties that allow its *attributes* to be defined:

- *readiness for usage* leads to **availability**;
- *service continuity* leads to **reliability**;
- non-occurrence of catastrophic consequences for the environment leads to **safety**;
- non-occurrence of unauthorised disclosure of information leads to **confidentiality**;
- non-occurrence of inadequate information alterations leads to **integrity**;
- ability to conduct repairs and introduce evolutions leads to **maintainability**.

Security is generally considered as the combination of confidentiality, integrity and availability [ITSEC], in particular relative to the authorised actions.

A **failure** of the system occurs when the delivered service deviates from implementing the system **function**, that is, from what the system *is intended for*. An **error** is that part of the system state that *may lead to a failure*: an error affecting the service is an indication of a failure occurring or which has occurred. The *adjudged or hypothesised cause* of an error is a **fault**.

Development of a dependable system requires the combined use of a set of methods that can be listed as follows:

- **fault prevention**: how to prevent the occurrence or introduction of faults;
- **fault tolerance**: how to provide a service implementing the system function despite faults;
- **fault removal**: how to reduce the presence (number, severity) of faults;
- **fault forecasting**: how to estimate the presence, creation and consequences of faults.

The notions that have been introduced can be listed under three main headings (as shown in Figure 1):

- **impairments** to dependability: faults, errors, failures; these are undesirable — but not unexpected — circumstances, causes or results of un-dependability (that can be

[1] The concepts as presented here are roughly equivalent to those presented in [Laprie 1995].

simply derived from the definition of dependability: trust can no longer, or will no longer, be put in the service delivered);

- the **means** for dependability: fault prevention, fault tolerance, fault removal, fault forecasting; these are the methods and techniques giving the system the ability to deliver a service conforming to the accomplishment of its function, and to place trust in this ability;
- **attributes** of dependability: availability, reliability, safety, confidentiality, integrity, maintainability: these enable a) expression of the properties expected from the system, and b) assessment of the quality of the service delivered, as resulting from the impairments and the means used to avoid them.

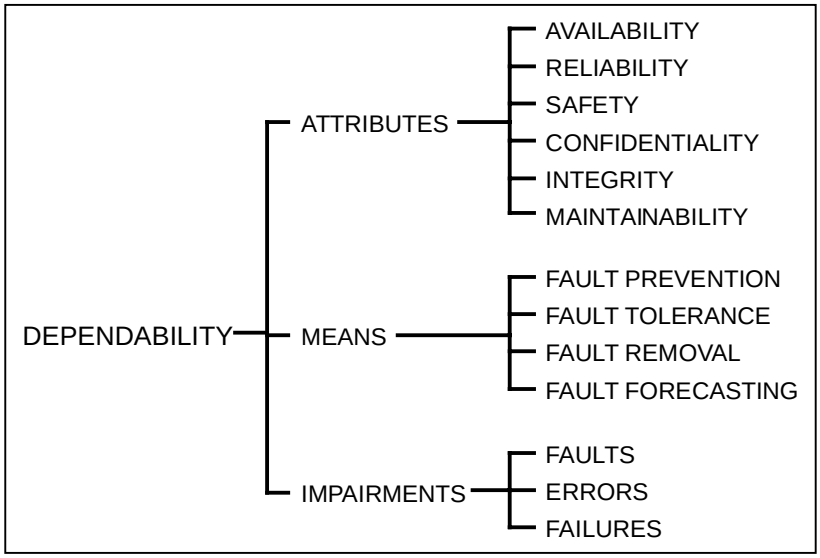


Figure 1 — The dependability tree

2.2 On the function, behaviour and structure of a system

So far, a **system** has — implicitly — been considered as a whole, emphasis being placed on behaviour as perceived from the outside. A definition according to this "black box" vision is: a system is an entity having interacted or interfered, interacting or interfering, or likely to interact or interfere, with other entities, that is, other systems. The latter make up or will make up the **environment** of the system considered.[2] A system user is a part of the environment that *interacts* with the system: a user provides inputs to the system and/or receives outputs from it. In other words, what distinguishes a user from the other parts of the environment is the fact that he uses *the service* delivered by the system.

As already pointed out in Section 2.1, the function of a system is what it is *intended for*. The **behaviour** of a system is what it *does*. What *enables it to do what it does* is its **structure.** [Ziegler 1976]Adopting the spirit of [Lee & Anderson 1990], a definition of a system from a structural point of view ("white box" or "glass box") is the following: a system is a set of

---

[2] a) Giving recursive definitions allows the relativity of the notion of system to be underlined according to the point of view considered: a system will not be looked at in the same way by a designer, its users and the maintenance teams.

b) Use of the past, present and future is intended to show that the system environment will change, particularly during the various phases of the life cycle. For example, the definition is broad enough to cover both the "programming environment" using during the development of the system, and the "physical environment" to which the system is subjected during its operational life.

components interconnected in order to interact; a **component** is another system, etc. Decomposition ends when a system is considered **atomic**: in other words, no subsequent decomposition is envisaged either by nature or because it is (currently) devoid of interest. The term "component" must be understood in its broad sense: layers of a system as well as intra-layer components; in addition, a component being itself a system encompasses the relationships between the components that make it up. A more conventional definition of the structure of a system is what the system *is*. This definition remains appropriate as long as dependability impairments are not considered and, therefore, the structure is considered frozen. However, as dependability impairments can be structural changes, or can cause or result from structural changes, a structure can have states.[3] Hence a definition of the notion of state: a **state** is a condition of being *relative to a set of circumstances*; this definition applies to the behaviour of a system as well as to its structure.[4]

Owing to its definition (the behaviour perceived by a user), the service delivered by a system is clearly an *abstraction* of the latter's behaviour. It is worth pointing out that this abstraction directly depends on the application for which the system is used. One example of this is the role played by time of this abstraction: time granularities of a system and of its users are usually different and vary according to the application concerned. In addition, the notion of service is not, of course, limited to outputs only but includes all interactions of interest to the user; for example, sensor scans are clearly part of the service expected from a monitoring system.

So far, we have used the singular for function and service. Usually, a system implements more than one function and delivers more than one service. Thus, function and service can be considered as composed of function elements and service elements. For clarity, we will use the plural — functions, services — when it is necessary or useful to make a distinction between several elements of function or service.

Given the preceding definition for the structure of a system, the notions of function and service naturally apply to components. This is particularly relevant in the design process when pre-existing hardware or software components are incorporated into a system: the designer is more interested in the function of the component or service it delivers, than its detailed (internal) behaviour.

The **specification** of the system, that is, an *agreed*[5] description of the function or service expected from the system, plays a pivotal role in dependability. Generally, the function or service is described or specified first in terms of what should be implemented or delivered according to the primary purpose of the system (for example, to carry out transactions, order or monitor a process, pilot a plane or guide a missile, etc.). With respect to security or safety systems, this description is usually completed by a statement of what should not occur (for example hazardous states that could cause a catastrophe or the disclosure of sensitive information). This latter description leads to the identification of additional functions the

---

[3] One can therefore say that a structure also features a behaviour, particularly, relative to dependability impairments, even if the pace of changes considered relative to, on the one hand, the user's requests and, on the other the dependability impairments, are — as should be noted — radically different.

[4] This definition is designed to lay the stress on a notion of state that depends directly on the phenomena and circumstances considered; for example: states relative to the information processing activities, states relative to the occurrence of failure, etc.

[5] An agreement is usually struck between two persons or groups of persons, physical or moral: the system vendor (in its broad sense: designer, manufacturer, seller, etc.) and its human users. The agreement may be explicit, e.g., when defining a new system to be built from scratch, or implicit, e.g., when purchasing an existing system with its specification and the user's manual, or when employing off-the-shelf systems.

system should implement to reduce the possibilities of what should not occur (e.g., identification of a user and verification of his rights).

In addition, the specification of these diverse functions can be:

- expressed according to various points of view or degrees of detail: specification of the needs, design specification, implementation specification, etc.,
- decomposed in accordance with the absence or presence of a failure; the first case relates to what is usually referred to as the *nominal* mode of operation and the second may deal with the so-called *degraded* mode of operation, if the remaining resources are no longer adequate for the nominal mode to be provided.

As a result, there exist several specifications, not one only, and a system can fail relative to one of them while still satisfying the others.

The expression of the functions of a system is an activity that is naturally initiated in the very early stages of a system development. However, generally, it is not limited to this phase of a system lifetime. In fact, experience has shown that the process of specifying the system functions has to be pursued throughout the system's lifetime as it is difficult to identify what is expected of it.

2.3 Human-made faults

Faults and their sources are highly diverse. Five main points of view can be considered to classify them. These are the phenomenological cause, nature, phase of creation or occurrence, situation relative to the system boundaries, and persistence [Laprie *et al.* 1995, chapter 1] (see Figure 2).

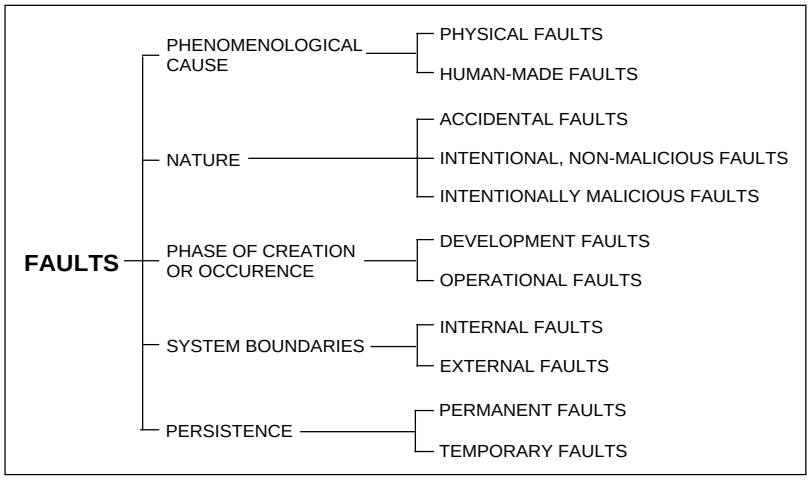


Figure 2 — Classes of elementary faults

Of particular interest to MAFTIA, are *human-made faults*, which correspond to four classes of combined faults:

- **design faults**, which are development faults, accidental or intentional with no malicious intent;
- **interaction faults**, which are external faults, accidental or intentional with no malicious intent;

- **malicious[6] logic**, which consists of malicious internal faults;
- **intrusions**, which are malicious, externally-induced, operational faults.

Some comments upon these classes of human made faults:

1 Intentional design faults with no malicious intent usually result from tradeoffs during the development, made with a concern for maintaining a suitable level of system performance or for facilitating system use, or even for economic reasons; these faults can be sources of security impairments in the form of hidden channels. Intentional interaction faults with no malicious intent can result from an operator attempting to address an unexpected event or deliberately acting in breach of procedures without realising the detrimental effects of his action. Generally, intentional faults performed without malicious intent are only identified as such after they have caused an unacceptable behaviour of the system, hence a failure.

2 Interaction faults are defined above as a class of human-made external faults that includes both accidental faults and intentional faults without malicious intent. These sub-classes should not be confused with the two error classes commonly considered for operators [Norman 1983]: intention errors (i.e., errors in the formulation of the interaction objective) and execution errors (i.e., errors in implementing these intentions).

3 Malicious logic covers development faults such as Trojan horses, trapdoors, logic or timing bombs, and operational faults (for the system considered) such as viruses and worms [Landwehr *et al.* 1994]. These faults include:

- a **logic bomb** is part of a program that remains dormant in the host system till a certain time or an event occurs, or certain conditions are met, triggering severe consequences for the host system;
- a **zombie** is a malicious program that can be triggered by an attacker in order to mount a coordinated attack;
- a **Trojan horse** is a program performing an illegitimate action while giving the impression of being legitimate; e.g., the illegitimate action can be the disclosure or modification of information (attack against confidentiality or integrity) or the triggering of a logic bomb;
- a **trapdoor** is a means of circumventing access control mechanisms; it is a flaw in the security system due to an accidental or intentional design fault (Trojan horse in particular);
- a **virus** is a program segment 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;
- a **worm** is an independent program that replicates itself and propagates without the users being aware of it; a worm can also carry a logic bomb.

2.4 Fault tolerance

Fault tolerance [Avizienis 1967] is carried out by error processing and by fault treatment [Anderson & Lee 1981]. **Error processing** is aimed at removing errors from the computational state, if possible before failure occurrence; **fault treatment** is aimed at preventing faults from being activated — again.

---

[6] Note that we use the term "malicious" in its standard dictionary sense of "desiring to harm others or to see others suffer" and *not* in the sense of "worst possible behaviour" that is sometimes used with respect to arbitrary faults.

Error processing can be carried out via three primitives:

- **error detection**, which enables an erroneous state to be identified as such;
- **damage assessment**, which aims to evaluate the extent of the damage[7] caused by the detected error, or by errors propagated before detection;
- **error recovery**, where an error-free state is substituted for the erroneous state; this substitution may take on three forms:
  - **backward recovery**, where transformation of the erroneous state consists of bringing the system back to a state already occupied prior to error occurrence; this involves the establishment of recovery points, which are points in time during the execution of a process for which the then current state may subsequently need to be restored;
  - **forward recovery**, where transformation of the erroneous state consists of finding a new state, from which the system can operate (frequently in a degraded mode);
  - **compensation**, where the erroneous state contains enough redundancy to enable its transformation into an error-free state.

When backward recovery or forward recovery are utilised, error detection must precede error recovery. These techniques are not antagonistic: a backward recovery may first be attempted; if the error persists, forward recovery may then be undertaken. In the latter case, the damage assessment must take place before undertaking recovery; damage assessment is not — in theory — necessary in the case of a backward recovery provided the mechanisms for implementing error recovery have not been affected [Anderson & Lee 1981].

The addition of error-detection mechanisms to the component's functional processing capabilities leads to the notion of **self-checking component**, for the hardware [Carter & Schneider 1968, Wakerly 1978, Nicolaïdis *et al.* 1989] or software [Yau & Cheung 1975, Laprie *et al.* 1990]. One of the main advantages of self-checking components is the possibility of clearly defining **error confinement domains** [Siewiorek & Johnson 1982]. When error compensation is carried out in a system made up of self-checking components partitioned into given classes of task execution, error recovery reduces to a switchover from a failed component to a non-failed one within the same class. On the other hand, error compensation may be applied systematically, even in the absence of errors, thereby providing **fault masking** (e.g., through a majority vote). Error detection is not, then, strictly speaking, needed to perform recovery. However, to avoid an undetected decrease in the redundancy available during a component failure, practical implementations of masking usually include an error-detection facility, which may in this case be initiated *after* recovery.

The operational time overhead (in terms of execution) needed for error processing may vary considerably according to the technique used:

- in the case of error recovery based on backward recovery or forward recovery, the time overhead is more important when an error occurs than when it does not. In the case of backward recovery, the time overhead consists in establishing recovery points and, therefore, in laying the groundwork for error processing;

---

[7] "Damage" refers here to error propagation *within* the system, i.e., before a failure has occurred (if failure has already occurred, recovery, and thus damage assessment, would no longer be worthwhile). In traditional fault-tolerance, damage assessment refers, for example, to finding out how many checkpoints to roll back to when doing backward recovery, or to finding out how many processes (might) have been affected by an error that has just been detected. With respect to malicious faults, damage assessment might be, for example, judging which files an intruder has modified so that they can be appropriately restored before someone needs to use them.

- in error compensation, the time overhead remains unchanged or almost the same whether or not there exists an error.[8]

In addition, error compensation is much faster than with a backward recovery and forward recovery owing to the much more important structural redundancy. This remark

- carries a certain weight in practice because it often conditions the choice of a fault-tolerance strategy relative to the time granularity of the system user;
- introduces a relationship between operational time overhead and structural redundancy. More generally, a redundant system always provides redundant behaviour, incurring at least some operational time overhead. The time overhead may be small enough not to be perceived by the user, which means only that the service is not redundant. An extreme form of time overhead is "time redundancy" (redundant behaviour obtained by repetition), which needs to be at least initiated by a structural redundancy, even in a limited form. Typically, the greater the structural redundancy, the lower the time overhead.

The first step in fault treatment is **fault diagnosis**, which consists of determining the causes of errors in terms of localisation and nature. Then, the steps needed to fulfil the main objective of fault treatment are carried out to prevent faults from being reactivated, hence **fault isolation**. To do so, the components deemed faulty are removed from the subsequent execution process[9]. If the system is no longer able to provide the same service as before, a **reconfiguration** may be envisaged by modifying the system structure so that fault-free components provide an adequate, although degraded, service. Reconfiguration may mean scrapping a number of tasks, or reallocating some of them to the remaining devices.

If it is thought that the fault has vanished after error processing or if its probability of recurrence is low enough, isolation becomes unnecessary. As long as fault isolation has not been undertaken, a fault is considered **soft**; undertaking isolation means that the fault is **hard**, or **solid**. At first sight, the notions of soft fault and hard fault may seem synonymous with that of temporary fault and permanent fault. Indeed, temporary faults can be tolerated without the need for fault treatment since error recovery should theoretically directly suppress the effects of a temporary fault, which will vanish unless a permanent fault is created by the propagation process. In fact the notions of soft fault and hard fault are useful, for the following reasons:

- distinguishing between a permanent fault and a temporary one is not easy and highly complex since a temporary fault vanishes after a certain amount of time, generally, before diagnosis is carried out and distinct classes of faults can give rise to similar errors; thus, the notion of soft or hard fault carries implicitly the subjectivity associated with these difficulties, including the fact that a fault can be considered soft following unsuccessful diagnosis;
- distinguishing between soft and hard faults makes it possible to take into consideration subtleties in the action modes of certain transient faults; for example, can a dormant fault due to the action of alpha particles, or of heavy ions in space, on memory elements (in the broad sense of the word, including flip-flops) be regarded a *temporary* fault*?* This fault is definitely a *soft* fault however.

The foregoing considerations apply to physical faults as well as design faults: the fault classes that can be tolerated in practice depend on the fault assumptions made in the design process, which are conditioned by the independence of redundancies relative to the fault creation and activation processes. An example is provided by considering tolerance of physical faults and tolerance of design faults. A (widely-used) method to attain fault tolerance is to perform

---

[8] In all cases, the time to update the system state tables increases the time overhead.

[9] Usually, removed faulty components can be repaired and re-inserted into the system; such curative maintenance can be considered as an ultimate form of fault-tolerance.

multiple computations through multiple channels. When tolerance of physical faults is foreseen, the channels may be identical, based on the assumption that hardware components fail independently. However, such an approach is not suitable for the tolerance of design faults. To tolerate design faults, the multiple channels have to provide *identical services* through *separate designs and implementations* [Elmendorf 1972, Randell 1975, Avizienis 1978], i.e., through **design diversity**. Design diversity is intended to tolerate permanent design faults. On the other hand, a backward recovery based error processing usually enables temporary design faults to be tolerated [Gray 1986].

An important aspect of co-ordinating the activities of multiple components is ensuring that the propagation of errors has no effect on fault-free components. This is particularly important when a given component must transmit a piece of information to other components. Typical examples of *information obtained from a single source* are local sensor data, the value of a local clock, the local perception of the state of the other components, etc. As a result, fault-free components must *agree* on how to use consistently the information obtained and, therefore, protect against possibly inconsistent failures (e.g., Byzantine agreement [Pease *et al.* 1980], atomic broadcast [Cristian *et al.* 1985], clock synchronisation [Lamport & Melliar-Smith 1985, Kopetz & Ochsenreiter 1987] or membership protocols [Cristian 1988]). It should be noted, however, that the unavoidable presence of structural redundancies in any fault-tolerant system requires a resource distribution at one level or another, leading to the persistence of the consensus problem. Geographically localised fault-tolerant systems may employ solutions to the agreement problem that would be deemed too costly in a "classical" distributed system of components communicating by messages (e.g. inter-stages [Lala 1986], multiple stages for interactive consistency [Frison & Wensley 1982]).

The knowledge of certain properties of the system may allow the required redundancy to be limited. Classical examples are given by regularities of a structural nature: error detecting and correcting codes [Peterson & Weldon 1972], robust data structures [Taylor *et al.* 1980], multiprocessors and networks [Pradhan 1986, Rennels 1986], algorithm-based fault tolerance [Huang & Abraham 1982]. The faults that can be tolerated are then dependent upon the properties considered since these properties are directly involved in the fault assumptions made during the design.

Warning users about the failure of a component is extremely important. This can be taken into consideration within the framework of exceptions [Melliar-Smith & Randell 1977, Cristian 1980, Anderson & Lee 1981]. *Exception handling* facilities provided in some languages may constitute a convenient way for implementing error recovery, especially forward recovery[10].

Fault tolerance is (also) a recursive concept; the mechanisms designed to tolerate faults must be protected against the faults likely to affect them. Examples are given by the replication of voters, self-checking controllers [Carter & Schneider 1968], through the notion of "stable" memory [Lampson 1981] in recovery data and programs.

Fault tolerance is not limited to accidental faults. Protection against intrusions has long relied on cryptography (e.g., see [Denning 1982] for an early overview). In particular, encryption can be viewed as a form of tolerance in that ciphered information can be inspected by an intruder without compromising its confidentiality, and authentication codes and digital signatures provide integrity against the same class of intruders. Secret sharing can be seen as error masking for confidentiality, too [Shamir 1979, Simmons 1991], and there are now many more types of cryptographic primitives and protocols (see, e.g., [Schneier 1996] for an informal overview). Certain error-detection mechanisms are designed for accidental as well as intentional faults (e.g., memory access protection techniques), and approaches have been put

[10] The term "exception", due to its origin of coping with exceptional situations — not only errors — should be used carefully in the framework of fault tolerance: it could appear as contradicting the view that fault tolerance is a natural attribute of computing systems, taken into consideration from the very initial design phases, and not an "exceptional" attribute.

forward to tolerate both intrusions and physical faults [Fray *et al.* 1986, Rabin 1989], and to tolerate malicious logic faults [Joseph & Avizienis 1988]. The MAFTIA project aims to follow this very approach, by building on this and other earlier work and extending it to the case of large distributed systems.

Chapter 3 Refinement of core concepts with respect to malicious faults

In this chapter we interpret 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 attributes

Dependability is defined in Section 2.1 as "that property of a computer system such that *reliance can justifiably be placed on the service it delivers*". According to the application requirements, the service may be requested to exhibit certain functional or non-functional properties, such as, for instance, accuracy of the computation results, respect of real-time deadlines, or other "quality-of-service" characteristics.

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[11] related to the information or service. Examples of such meta-information are:

- time of a service delivery, or of creation, modification or destruction of an item of information;
- identity of the person who has realised 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 of the 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;
- etc.

For example, *accountability* [CEN 13608-1, ISO 7498-2, Trouessin 2000] corresponds to the availability and integrity of a set of meta-information about the existence of an operation, the identity of the person who has realised the operation, the time of the operation, etc. *Anonymity* is the confidentiality of the identity of the person, for instance, who realised (or did not realise) an operation. *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.

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

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

user who has performed a particular operation, or sent a particular message, or received the message, 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 integrity of some meta-information, such as creator identity (and possibly time of creation) for non-repudiation of origin, or such as reception and receiver identity for non-repudiation of reception.

It is conjectured that all security properties can be expressed by the availability, integrity and confidentiality properties applied to information and meta-information.

3.2 Security policy

Meaningful discussion of security-related topics often requires reference to a *security policy*. Unfortunately, there are many different senses of the term applied to many different levels of abstraction.

At the highest level, the (high-level) security policy describes the system, the properties it should have, and who (at least in title) is responsible for what; it includes the *security* portion of the *specification* of the system's *function*.

Systems are recursively composed of subsystems; a system's specification recursively determines the specifications of its subsystems.

Eventually, through recursive refinement, specialisation, and implementation, the system's *behaviour* is determined. This process includes the creation of guidelines, processes, site descriptions, practices, specifications, mechanisms, implementations and configurations.

At the lowest level, we can view the system as a (very large) finite state machine (Figure 3) in which the black state is disallowed by the security policy (i.e., the failure-state). The pale states designate normal states, whereas the hashed states represent error states that may lead the system to the failure-state.

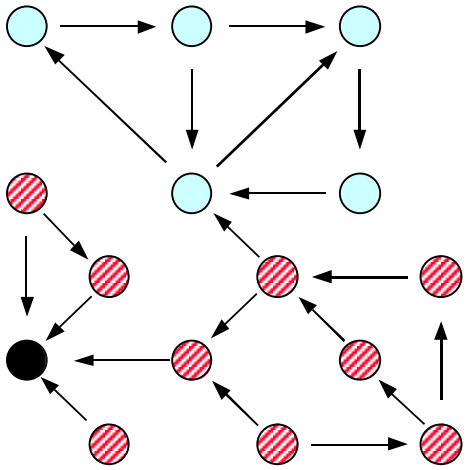


Figure 3 — Low level security policy

At this level, the security policy is the collection of rules according to which the system's security state should evolve. It is usual to separate security policies from functional properties,

in which case the *security state* can be viewed as a (very large) matrix of subject privileges on object operations. The security policy rules specify the legitimate evolutions of this state.

For our purposes, then, we will define a security policy as:

1) the security properties that are to be fulfilled by the system (high-level view);
2) the rules according to which the system security state may evolve (low-level views).

The properties specified by the security policy are defined in terms of the security attributes (cf. Section 3.1) that are required for the services delivered to the various stakeholders of the system. A **security failure** occurs when a property of the intended[12] security policy is violated. Such a failure may occur in two ways:

- the security policy is inconsistent, for which three cases can be distinguished:
  - the specified security properties are antagonistic
  - the specified rules are contradictory
  - the rules and properties are mutually inconsistent, including the case of rules being incomplete
- the rules are violated (e.g., due to an intrusion or other fault) so the transitions between states are different to those of the state machine inferred from the rules.

In between the high and low-level views, one can view the system with different levels of abstraction. For our purposes, it is important to think of the security policy as being the recursively structured complete collection of these views. Doing so corresponds naturally with the recursive structure of dependability practices and provides a framework in which to discuss security as a whole.

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 2.1, and then elaborate on potential causes of security failures.

3.3.1 Causal chain of impairments

In Section 2.1, 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. Figure 2, page 6) that could give rise to the same detectable impairment.

[12] We add the adjective "intended" to cater for the case where the security policy is incorrectly specified.

- 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 4, where the dotted lines represent a "system boundary", at the considered level of decomposition or abstraction.

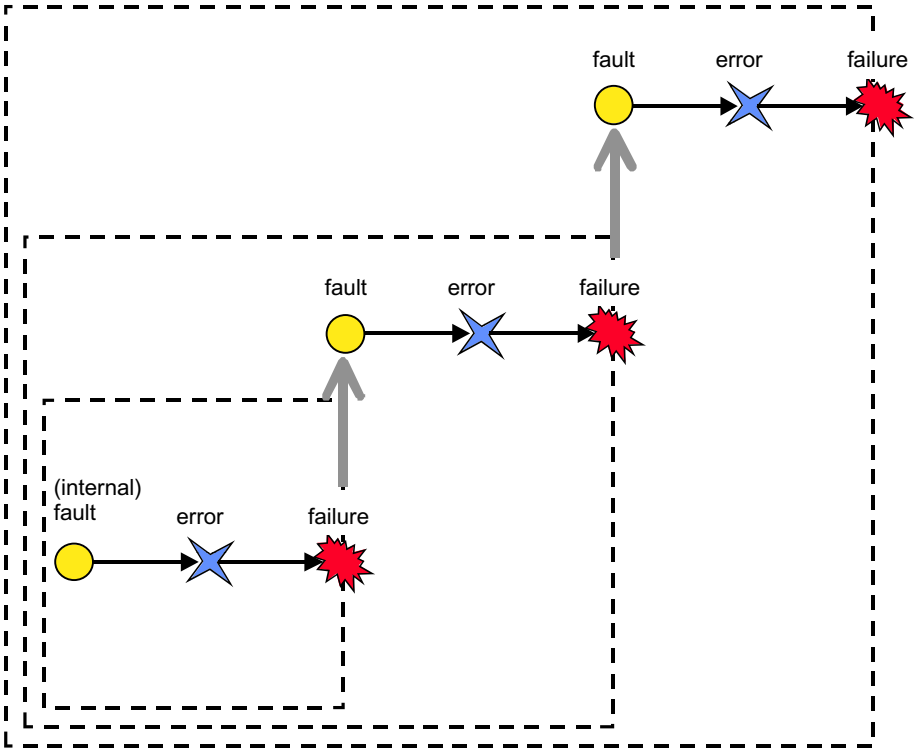


Figure 4 — Hierarchical causal chain of impairments

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

3.3.2 Intrusion, attack and vulnerability

An intrusion was defined in Section 2.3 to be a malicious, externally-induced, operational fault. 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 protective "shield" around the system or resource targeted by the attack is 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 5):

1. A malicious act or *attack* that attempts to exploit a weakness in the system,

2. At least one weakness, flaw or *vulnerability*, which is an accidental fault, or a malicious or non-malicious intentional fault, in the requirements, the specification, the design or the configuration of the system, or in the way it is used.

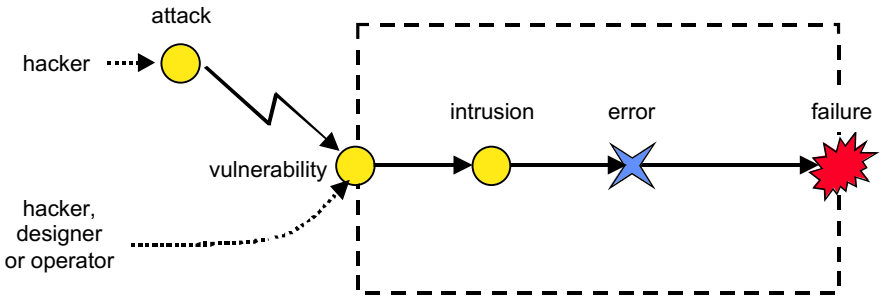


Figure 5 — Intrusion as a composite fault

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 authority (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[13] 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).

Let us now return to the notion of a hierarchical causal chain of impairments as represented by Figure 4, page 16. 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

[13] DDoS: Distributed Denial of Service.

is clearly an intrusion as seen from the containing system (Figure 6). 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.

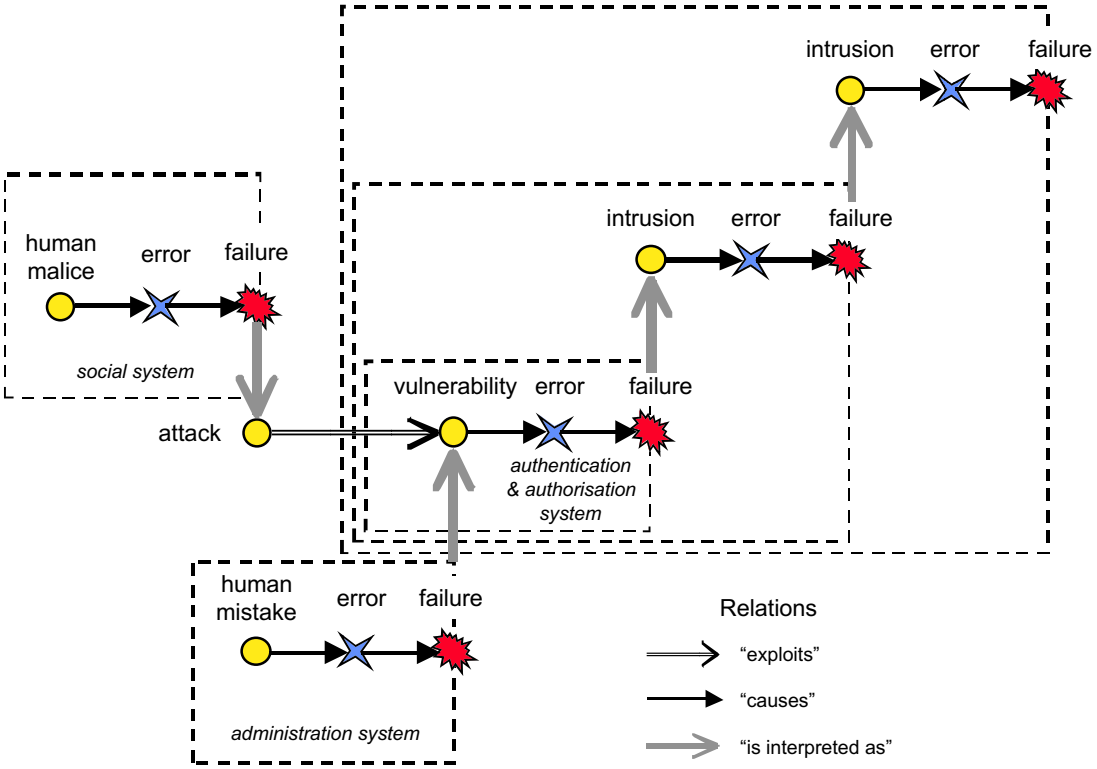


Figure 6 — Attack, vulnerability and intrusion in a hierarchical causal chain

Moreover, depending on the adopted viewpoint at a given level, the intrusion may also be viewed as either an attack or a vulnerability[DP28]. Indeed, the intrusion may result in a new vulnerability (the hacker exploits his successful attack to place a trapdoor or a zombie) or manifest itself as a further attack (the hacker directly exploits his successful attack in order to proceed towards his final goal).

Figure 6 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.

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 liable to lead to a security failure:

attack – a malicious interaction *fault* aiming to intentionally violate one or more security properties; an *intrusion* attempt.

vulnerability - an accidental fault, or a malicious or non-malicious intentional fault, in the requirements, the specification, the design or the configuration of the system, or in the way it is used, 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, essentially design or configuration faults (e.g., coding faults allowing program stack overflow, files with root setuid in UNIX, naïve passwords, unprotected TCP/IP ports). *Attacks* are malicious interaction faults that attempt to activate one or more of those 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 to the hacker). 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.

3.3.3 Theft and abuse of privilege

In the previous section, we referred 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[14]. 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 security policy.
- **abuse of privilege**: a misfeasance, i.e., an improper use of authorised operations.

These two notions are illustrated by Figure 7, 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*).

[14] 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*.

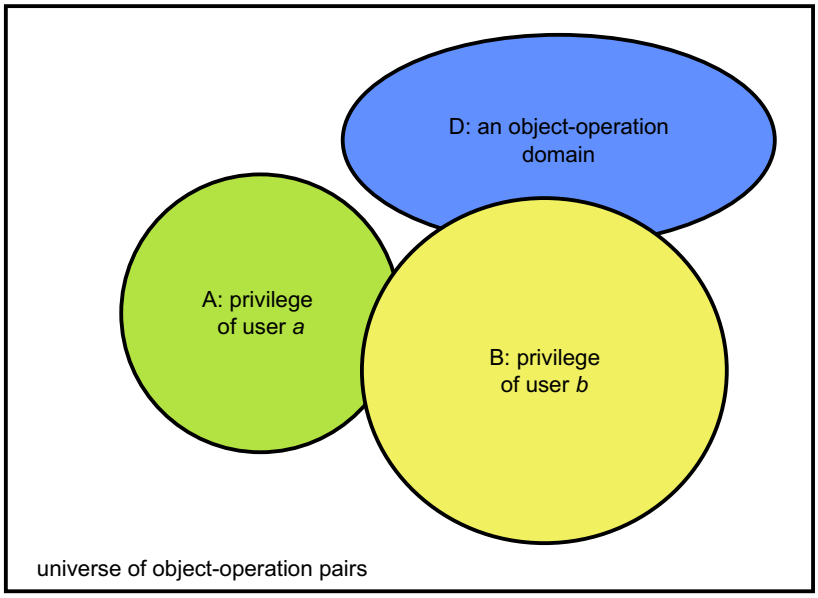


Figure 7 — Outsider (user a) vs. insider (user b) with respect to domain D

In Figure 7, 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 intrude on 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 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.

3.3.4 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[15];

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

[15] 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.

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 8).

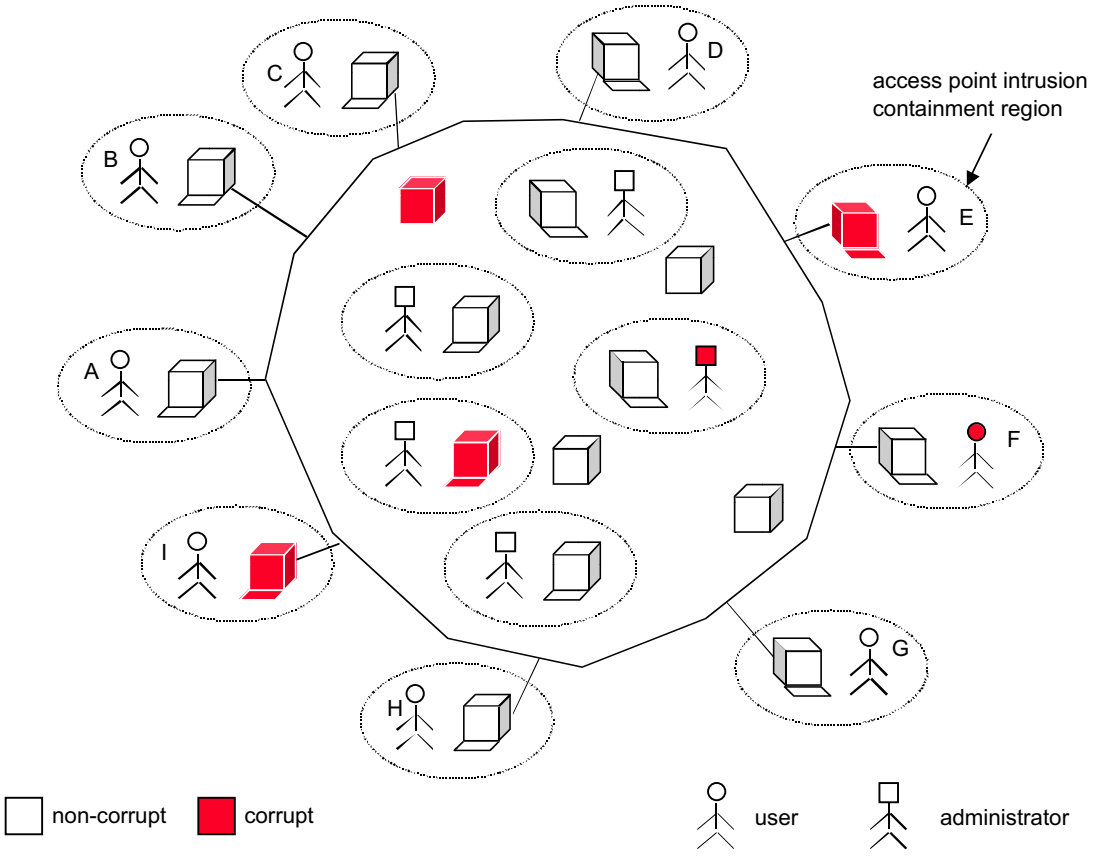


Figure 8 — Corrupt vs. non-corrupt access points and users

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 8).[16] 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 8). Indeed, from the security viewpoint, a user 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 8. For non-corrupt users of non-corrupt access points, an intrusion-tolerant system

[16] 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.

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 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*, *vulnerability* and *intrusion* with fault, and applying the definitions given in Section 2.1 we can obtain *a priori* twelve methods for ensuring or assessing security (Table 1). However, not all of these twelve methods are distinguishable or indeed meaningful.

We in fact obtain seven meaningful methods, which are presented in the following subsections.

Table 1 — Classification of security methods

		Attack	Vulnerability	Intrusion
Prevention:	how to prevent the occurrence or introduction of…	deterrence, laws, social pressure, secret service…	semi-formal and formal specification, rigorous design and management…	firewalls, authentication, authorisation… + {attack prevention, vulnerability prevention}
Tolerance:	how to provide a service capable of implementing the system function despite…	= {vulnerability prevention, vulnerability removal, intrusion tolerance}	≡ intrusion tolerance	error detection & recovery, fault masking, intrusion detection, fault treatment
Removal:	how to reduce the presence (number, severity) of…	not applicable	formal proof, model-checking, inspection, test…	not applicable
Forecasting:	how to estimate the creation and consequences of…	intelligence gathering, threat assessment…	assessment of: presence of vulnerabilities, exploitation difficulty, potential consequences…	= {vulnerability forecasting, attack forecasting}

3.4.1 Fault prevention

In Section 2.1, 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: how to prevent the occurrence of attacks;

This includes deterrence measures such as social pressure, laws and their enforcement.

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).

intrusion prevention: how to prevent the occurrence of intrusions;

This includes technical measures such as authentication, authorisation and firewalls, as well as attack and vulnerability prevention (see above).

3.4.2 Fault tolerance

In Section 2.1, fault tolerance is defined as "how to provide a service implementing the system function despite 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*.

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

intrusion tolerance: how to provide a service implementing the system function despite 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 2.1, fault removal is defined as "how to reduce the presence (number/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 often called corrective maintenance.

Equating *attack*, *vulnerability* and *intrusion* with fault appears to be meaningful only for vulnerability removal. Indeed, fault removal refers to verification methods (including testing) aimed at finding *internal* faults, so it is difficult to find any significance for the notions of intrusion removal or attack removal. Although countermeasures might be thought of as removing an attack (or maybe the attacker!), we believe these are better classified as a form of intrusion *tolerance* since, like maintenance or other fault treatment actions, they aim to maintain or to restore the ability of the system to fulfil its function.

Hence, we obtain one distinguishable set of fault-removal methods:

vulnerability removal: how to reduce the presence (number, severity) of vulnerabilities;

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, applying a security patch, withdrawing a given service, changing a password, etc.

3.4.4 Fault forecasting

In Section 2.1, fault forecasting is defined as "methods and techniques aimed at estimating the present number, the future incidence, and the consequences of faults". Equating *attack*, *vulnerability* and *intrusion* with fault, we obtain two clearly distinguishable sets of fault-forecasting methods:

attack forecasting: how to estimate the presence, creation and consequences of attacks.

This includes intelligence gathering, threat assessment and attack warning

vulnerability forecasting: how to estimate the presence, creation and 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 take advantage of them.

Security risk analysis can be viewed as a combination of both attack and vulnerability forecasting.

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.

* * *

Figure 9 illustrates the notions of fault prevention, tolerance and removal in the context of attacks, vulnerabilities and intrusions.

Note that:

- All procedures and mechanisms aimed at preventing or tolerating faults can be imperfect, i.e., they may contain faults that one should aim to remove.
- Faults in intrusion-prevention mechanisms constitute vulnerabilities that may be exploited by an attacker.
- Fault-tolerance techniques in general aim to prevent errors from propagating to the service interface, which would result in a failure.
- Intrusion-tolerance techniques in particular aim to prevent errors caused by intrusions from leading to failure; but it should not be forgotten that errors might be caused by faults other than intrusions.

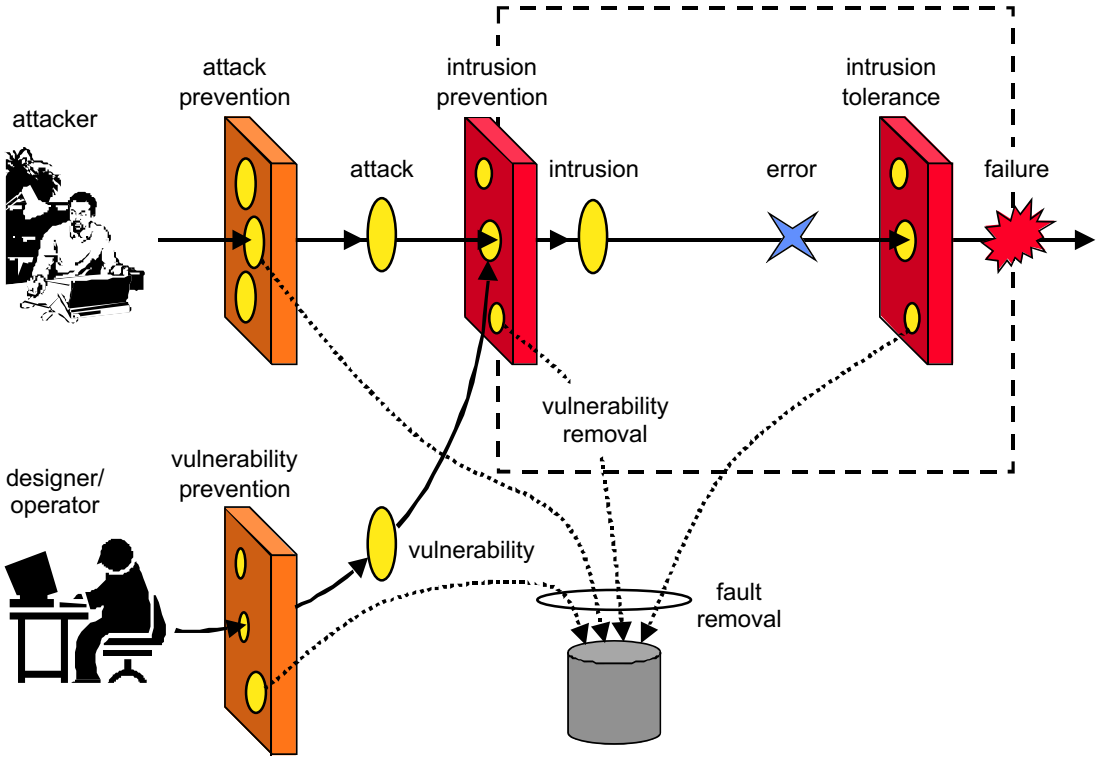


Figure 9 — Fault prevention, tolerance and removal

3.5 On the nature of trust

The notions of "trust" 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 "trustworthy" are a generalization of the NSA notions, and are based on the normal English sense of these words, which relate strongly to the words "depend" and "dependable" but more specifically to security issues. A component is "trusted" if something else "trusts" it. Here, "trust" means that it is assumed by the user of the component that some particular security specification will be adhered to, so that user does not have to allow for failures to adhere to this specification. A "trustworthy" component is a component that has been shown to be deserving of the trust placed in it. A trustworthy component from a security point of view is dependable with respect to its security properties. This can be achieved through the techniques of fault prevention, fault tolerance, fault removal, and fault forecasting.

More particularly, some of the intrusion-tolerance strategies adopted within MAFTIA rely upon some hardware elements being "tamper-proof"[17]. For example, the implementation of the authorisation service uses Java Cards to store private keys used to verify capabilities. Since we assume that these are tamper-proof, we argue that they are trustworthy in the sense that they will not reveal these keys to an unauthorised party.

[17] The coverage of the "tamper-proof" assumption may of course not be perfect. This is reflected in the term "tamper-resistant" sometimes used to describe hardware that is intended to be tamper-proof, but which cannot be guaranteed to be so in the current state of the art [Anderson 2001] [Akkar *et al.,* Weingart 2000].

Chapter 4 Intrusion tolerance

This chapter focuses on one of the seven 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 Chapter 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 Chapter 2 (Section 2.1), 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 lead to a failure*.

Whereas the definition of security failure is naturally derived from loss of confidentiality, integrity or availability (as defined in the properties of the considered security policy), there is currently no agreed definition of what might constitute an *error* from the security viewpoint. However, current literature refers to "intrusion detection" which, from the dependability concept viewpoint, might lead one to equate intrusion with "error", rather than "fault"[18]. 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[19] 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)."

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 treatment facility);
- error detection (as a preliminary to automatic error recovery), or error detection and on-line fault diagnosis (as a preliminary to automatic fault treatment).

Further confusion is introduced by the opposition in [Halme & Bauer] between a "manually reviewed IDS"[20] (called a passive IDS in [Debar *et al.* 1999]) and "Intrusion Countermeasure Equipment (ICE)" or "autonomously acting IDS" (sic) (called an active IDS in [Debar *et al.*

---

[18] 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.

[19] SSO: System Security Officer.

[20] IDS: Intrusion Detection System.

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".

However, 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 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 10 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.

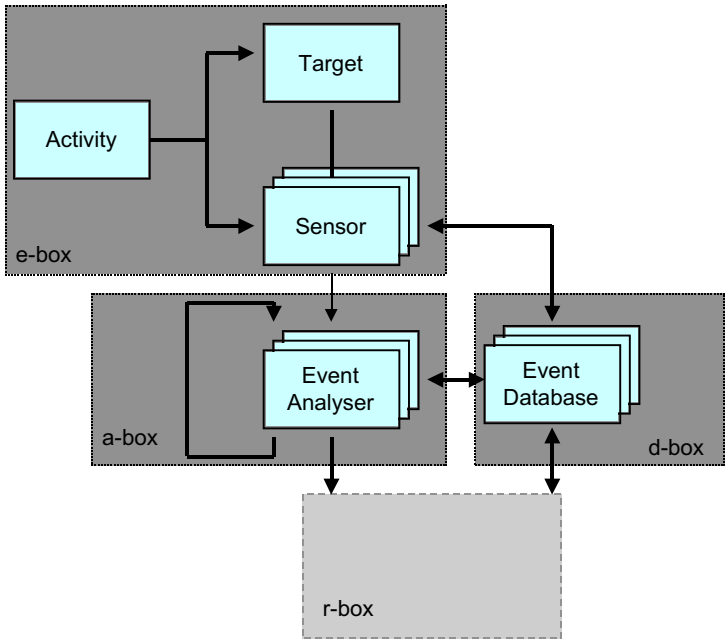


Figure 10 — Intrusion-detection system components

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 11).

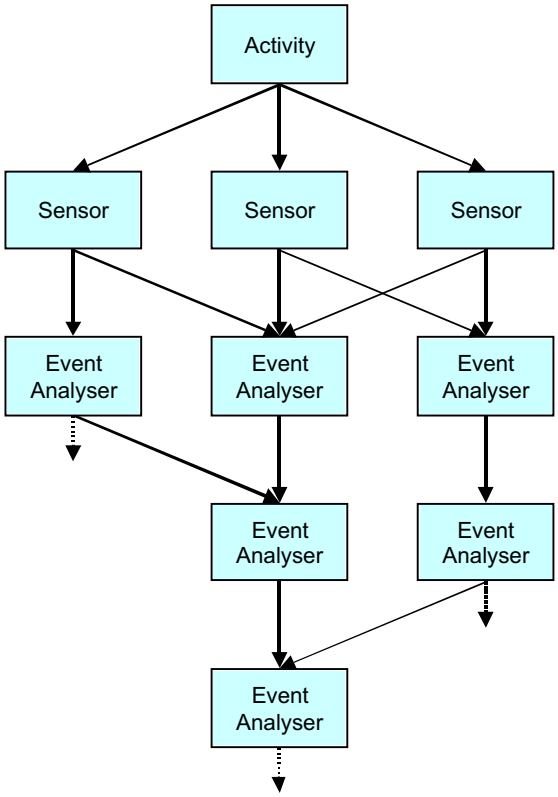


Figure 11 — Cascaded intrusion-detection topology

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:

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

Target is the name of the component that we are trying to monitor. We assume that the activity has some channel to the target.

Sensor is the component of the system collecting raw data (e.g., a sniffer or an audit log).

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 vulnerabilitiesprovided 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 will 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 11).

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

The *event database* is to provide 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 10 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].

4.3 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 2.4). Those concepts make a distinction between: (a) *error processing*, aimed at preventing errors from leading to (catastrophic) failure, and (b) *fault treatment*, aimed at preventing the recurrence of errors.

4.3.1 Error processing

In the core dependability concepts, error processing covers the set of techniques aimed at removing errors from the computational state, if possible, before the occurrence of a failure. Section 2.4 identifies three error-processing primitives: error detection, damage assessment and error recovery, which we examine successively in the context of intrusion-tolerance.

4.3.1.1 Error detection

Error detection is a necessary preliminary to achieving backward or forward 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 recovery method

employed (if any), error detection is necessary if subsequent fault treatment or curative maintenance actions are to be undertaken.

By definition, error-detection techniques (and indeed, error processing techniques in general) 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, and this for at least two reasons:

- Determination of whether or not the cause of error has malicious nature 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).

To detect errors that might be caused by intrusions, the relevance of malice is simply in setting requirements. If we are able to dependably detect errors caused by malicious faults, then we are implicitly able to dependably detect errors caused by non-malicious faults. The converse is not true.

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 12)[21]:

- 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.

[21] [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”.

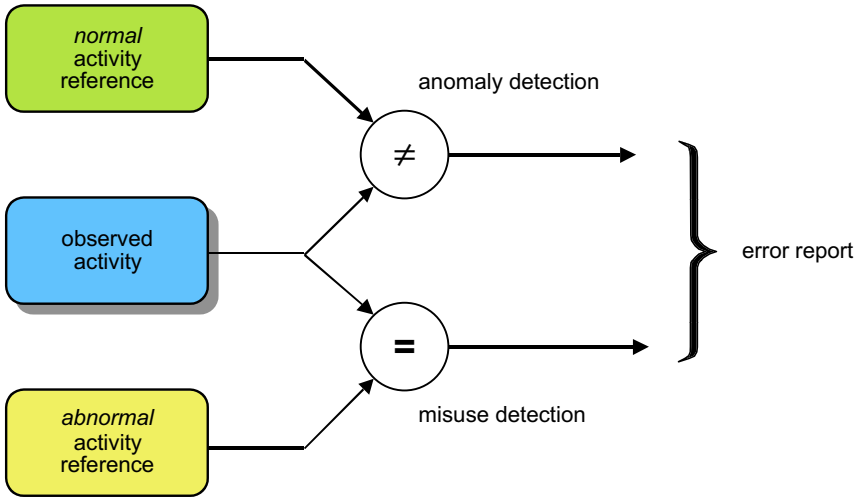


Figure 12 — Detection paradigms

The rules contained within the system's security policy (cf. Section 3.2) provide an important reference regarding what observed activities should be considered as erroneous from a security viewpoint[22]. 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 4.3.1.3 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. Indeed, this is particularly so when detecting errors caused by intrusions. 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). In intrusion-detection systems, the degree of *im*perfection is measured in terms of the rate of so-called false negatives and false positives, defined as:

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.

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

[22] Note, however, that our definition of a security *failure* (cf. Section 3.2) is given in terms of the security *properties* required by the security policy.

[Avizienis *et al.* 2001], such built-in self-test procedures are termed *pre-emptive error detection*, whereas the mechanisms considered till now are termed *concurrent 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 diagnosis, rather than a trigger for automatic recovery[23]. Indeed, vulnerability-scanning and configuration-checking are also useful tools for carrying out preventive maintenance.

4.3.1.2 Damage assessment

Damage assessment is an error processing primitive aimed at evaluating the extent of error propagation before attempting recovery. In traditional fault-tolerance, damage assessment refers, for example, to finding out how many checkpoints to roll back to when doing backward recovery, or to finding out how many processes (might) have been affected by an error that has just been detected. In intrusion-tolerance, damage assessment might be, for example, judging which files an intruder has modified so that they can be appropriately restored before someone needs to use them. Vulnerability-scanning also comes into play here, since it may be triggered to seek out maliciously-implanted vulnerabilities.

4.3.1.3 Error recovery

The core dependability concepts of Section 2.4 distinguish three forms of error recovery:

- **Backward recovery**: state transformation is carried out by bringing the system back to a previously occupied state, for which a copy (a recovery point) has been previously saved.
- **Forward 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 recovery for intrusion-tolerance, examples of each form of recovery include:

- **Backward recovery**:
  - Operating system re-installation
  - TCP/IP connection resets
  - System reboots and process re-initialisation
  - Software downgrades
- **Forward recovery**:
  - In threshold-cryptography, replacement of compromised key shares
  - Putting the system into a diminished operation, presumably safe, mode
  - Software upgrades (supposing that an upgrade is available on-line)
- **Compensation**:
  - Voting mechanisms
  - Fragmentation-Redundancy-Scattering
  - Sensor correlation

[23] In other situations, pr-emptive error detection may be followed by automatic recovery. Examples include memory-scrubbing, software rejuvenation, etc. [Avizienis *et al.* 2001].

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

4.3.2 Fault treatment

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

Here, we assume that whereas error processing is carried out entirely automatically, fault treatment might be, at least partly, manual, e.g., with the aid of a system security officer or administrator.

Section 2.4 identifies three fault treatment primitives: fault diagnosis, fault isolation, and (system) reconfiguration.

4.3.2.1 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.).

4.3.2.2 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.

4.3.2.3 System reconfiguration

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 terms of intrusions, possible reconfiguration actions include:

- 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.

Actions pertaining to corrective maintenance might be:

- 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 13.

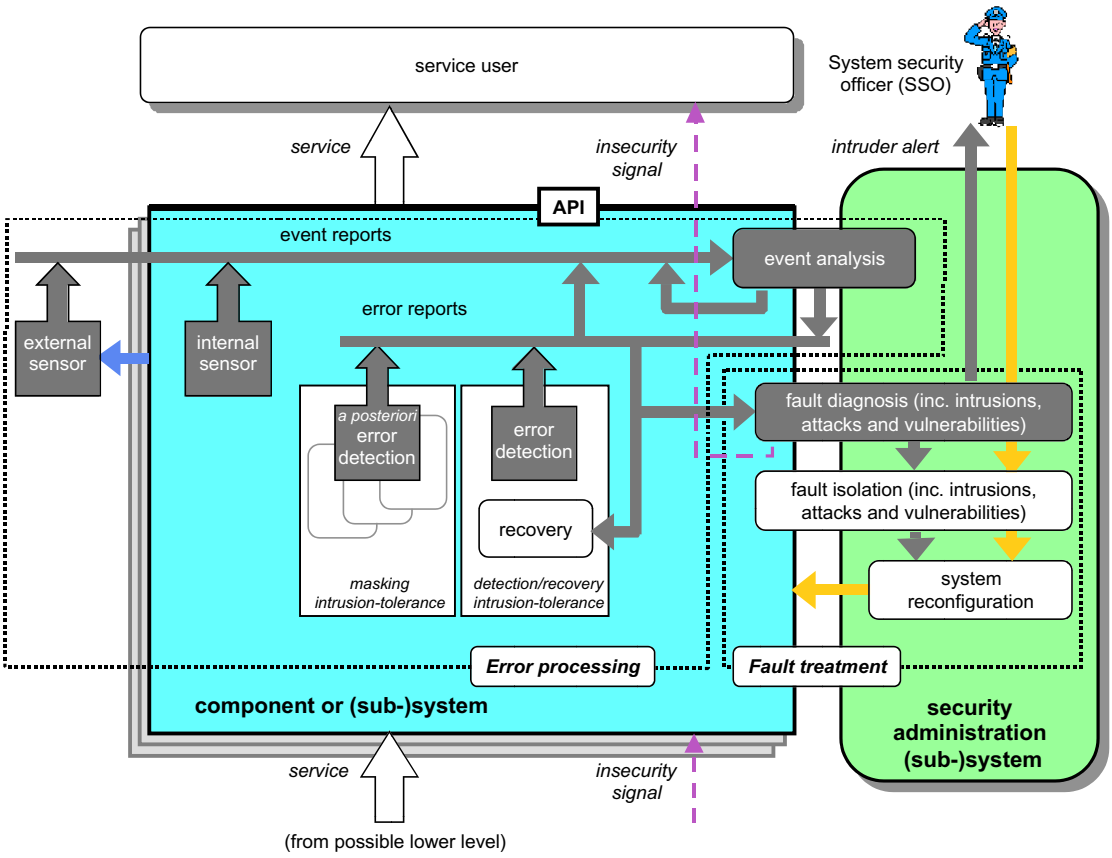


Figure 13 — Integrated intrusion-tolerance framework

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 13 in terms of error processing and fault treatment.

4.4.1 Error processing

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 treatment facilities, which may carry out further analysis toward understanding the root causes of detected errors (fault diagnosis), and then act thereon (fault isolation and system reconfiguration).

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 13. 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 (backward or forward) recovery, or intrusion-masking (cf. Section 4.3.1.3). In MAFTIA, particular attention is being paid 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, Rabin 1989, Deswarte *et al.* 1991, Fabre *et al.* 1994]. Possible applications of this approach include services based on *trustworthy* trusted third parties such as those described in MAFTIA deliverables D26 and D27 [Abghour *et al.* 2001, Cachin 2001]:

- 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 treatment facilities. Intrusion-tolerant components are thus a particular kind of internally-monitored components.

4.4.2 Fault treatment

The fault-treatment facilities include the means for diagnosing and isolating faults (including intrusions, attacks and vulnerabilities), and for automatic or manual system reconfiguration (cf. Section 4.3.2). 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 13 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 13.

4.4.3 An illustrative example

As an example of how intrusion-detection and intrusion-tolerance fit together, Figure 14 shows a much simplified, unfolded interpretation of Figure 13.

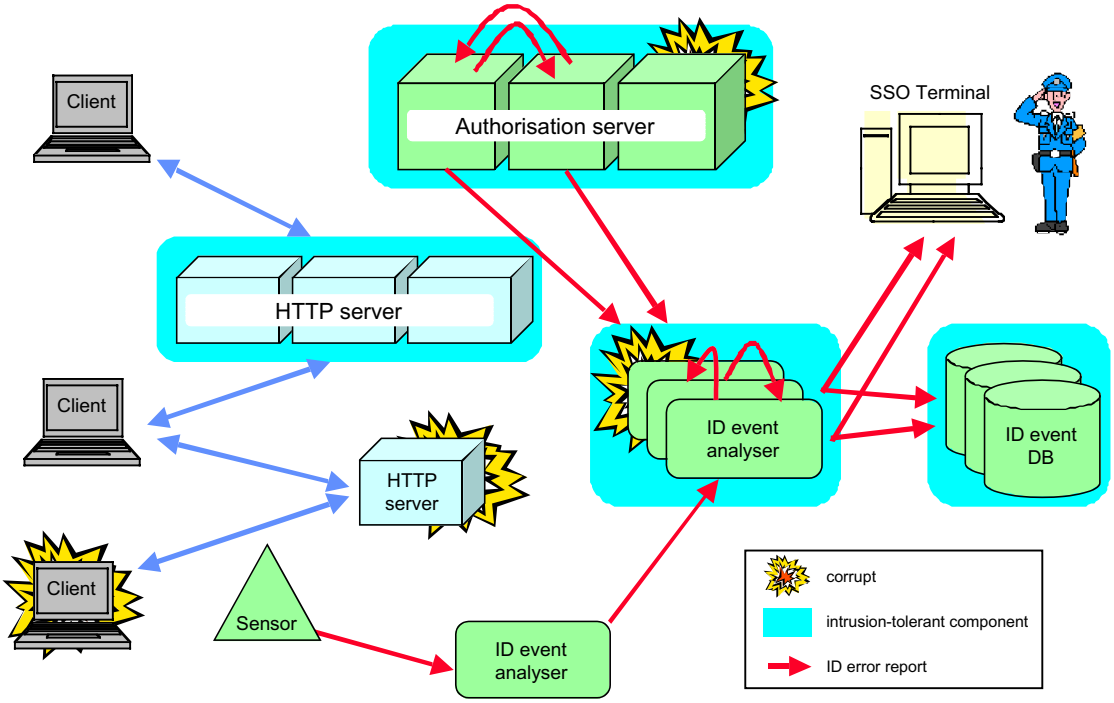


Figure 14 — Relationship between intrusion-detection and intrusion-tolerance

Figure 14 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 11, page 30) 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 Models and assumptions

5.1.1 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* assumption

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 extremely 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, Verissimo *et al.* 1997].

5.1.2 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 5, page 17), 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. Seven such security methods were defined in Section 3.4.

5.1.3 Enforcing hybrid failure assumptions

The second step is the enforcement of hybrid failure assumptions, that is, where different components are assumed to exhibit different faulty behaviours. 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.

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 vulnerability prevention and removal, attack and intrusion prevention, 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 3.5). 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 (or fail-uncontrolled) and fail-controlled 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.

5.1.4 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.

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[25] (cf. Section 3.4), can subsequently be used in the construction of fault-tolerant protocols under hybrid failure assumptions.

Note that the soundness of our approach does not depend on our making possibly naïve assumptions about what a hacker can or cannot do. 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 finally tolerating any intrusions 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 security kernels around components we need to be trusted. Such security kernels are simple enough to be built and plausibly shown to be correct. In other words, we consider them to be "trustworthy" in the sense discussed in Section 3.5. 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 two instances of such security kernels that we describe in more detail in the subsequent architecture overview (see Section 5.2.4). The first is a local security kernel, based on Java Cards, which is designed to assist the crucial steps of the execution of services and applications. The second is a distributed security kernel (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 classical prevention-based approaches (e.g., Reference Monitors), the security kernel does not mediate all accesses to resources and operations. In our approach, protocols run in an untrusted environment, local participants only trust interactions with the (trusted) security kernels, single components can be corrupted, and correct service provision is built on distributed fault-tolerance mechanisms, for example through agreement and replication amongst collections of participants in several hosts.

The local security kernel 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 security kernel 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.

5.1.5 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.

---

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

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 security kernels 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*, as long as there are a total number of participants $n > 3f$.

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.

5.1.6 Synchrony models

Research in distributed systems algorithms has traditionally been based on one of two canonical models: fully asynchronous and fully synchronous models [Verissimo & Raynal 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[26]

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 (MAFTIA deliverable D26 [Cachin 2001]). 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 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

[26] **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.

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.

5.1.7 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. 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; 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 5.1.4, the TTCB is built as a distributed and synchronous security kernel, 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. .

An interesting observation is that a partially synchronous system with a timing failure detector encompasses the entire spectrum of partial synchrony, from fully asynchronous to fully synchronous. In consequence, MAFTIA protocols can have an incremental degree of asynchrony, and ultimately, time-free protocols can be built that rely on accurate timed failure detectors in the TTCB, exclusively used to generate and control timeouts in a trusted way. *Trusted* should be read both from the viewpoint of full synchrony, and of tamper-proofness: it is impossible to have accurate timeout-based failure detectors in asynchronous systems, and/or systems subject to the manipulation of a malicious attacker.

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 is an architectural artefact supporting the construction and trusted execution of intrusion-tolerant protocols and applications running under a partially synchronous model.

5.1.8 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 [Cachin *et al.* 2000b]. 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.

5.1.9 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.

5.2 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.

5.2.1 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].

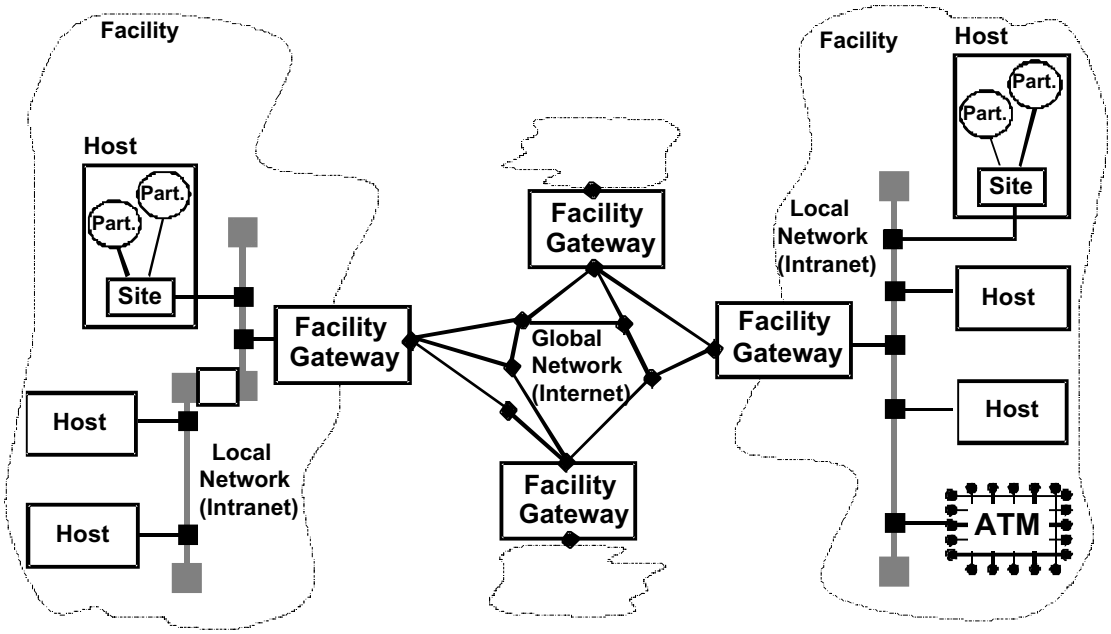


Figure 15 — Two-tier WAN-of-LANs

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 15: *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.

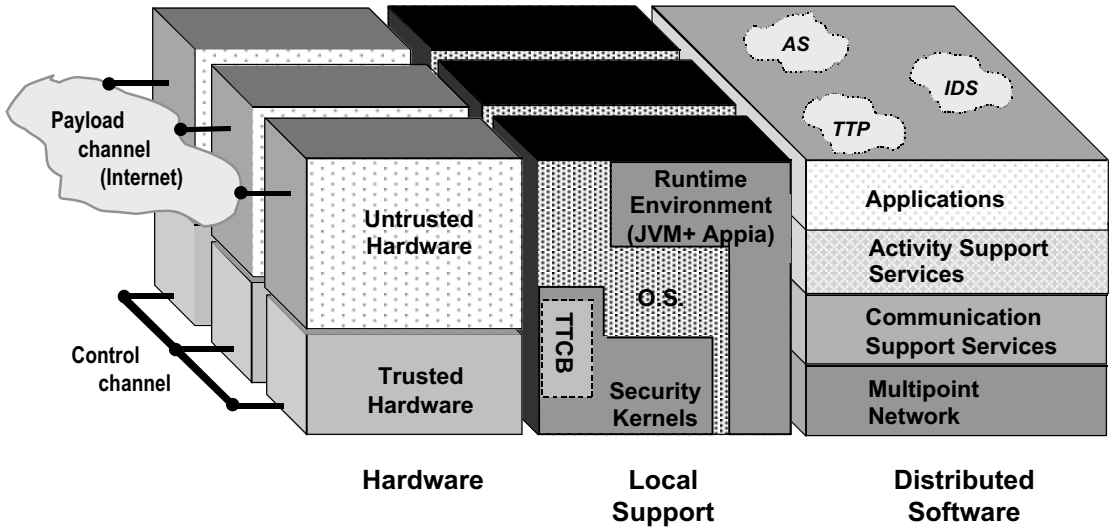
The WAN-of-LANs view we have just presented can be recursively applied, in order to represent very-large-scale organisations. On an intra-facility level, further hierarchies, namely those already deriving from hierarchical organisation of sub-networks and domains, are not precluded. On an intra-organisation (multi-facility) level, the topology depicted in Figure 15 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 15). 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.

5.2.2 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 Section 5.1:

- 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 3.5). 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 *security kernel*. This is the way in which MAFTIA endorses the notion of a fail-controlled subsystem trusted to execute a few functions correctly, albeit immersed in an environment subjected to malicious faults. The use of trusted hardware serves to substantiate this assumption with high coverage.
- 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.
- 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.



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

Figure 16 — MAFTIA architecture dimensions

The MAFTIA architecture can be depicted in at least three different dimensions (see Figure 16). First, there is the host and networking *hardware* device and their topology, briefly discussed earlier, which 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 MAFTIA deliverables D3 [Alessandri 2001], D23 [Neves & Veríssimo 2001], D26 [Cachin 2001], D27 [Abghour *et al.* 2001]. In the remainder of this section, we discuss in a little more detail the hardware, the local support, and the middleware.

5.2.3 Hardware

We assume that the hardware in individual MAFTIA hosts is untrusted in general. However (see Figure 16) some hosts may have pieces of hardware that are trusted in the sense of being regarded as tamper-proof (see Section 3.5), 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*. A common accessory in the PC family, it 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.

5.2.4 Local support

The local support dimension of the architecture (see Figure 16) 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 security kernels — the Java Card based module, and the Trusted Timely Computing Base (TTCB).

A security kernel is trusted from the viewpoints of correctness of its operation, and of intrusion prevention: the kernel provides correct functions in a fault free situation, and cannot be intruded upon. 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 it protects and any attempt to interact with them (it is always invoked)
- Shielding: it must be shielded (tamper-proof) from any contamination from the outside (errors propagating from the rest of the system, malicious attacks)
- Validation: it must be verifiable, in order to ensure very high coverage of its properties.

5.2.4.1 Java Card security kernel

This security kernel is mainly used to support authorisation services, where 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 security kernel controls all accesses to local objects by checking if each request carries an authorisation for the access. This authorisation may have been delivered by the authorisation server, if the access is an access to a persistent object, or by the security kernel itself, if the access is an access to a local transient object.

The security kernel 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). Trusting the security kernel has the following meaning: it is not feasible to subvert a security kernel, but it may be possible to interfere in its interaction with software components through the JVM. In this case however, we consider that local host as having been compromised, but we trust that the security kernel enforces error confinement. In distributed fault-tolerance terms, 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.

5.2.4.2 Trusted timely computing base

The distributed security kernel (TTCB) is 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 security kernel 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) inter-module communication.

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). Trusting the TTCB security kernel has a meaning similar to the Java Card security kernel: it is not feasible to subvert the TTCB, but it may be possible to interfere in its interaction with software components through the JVM. In similar terms, 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. In case that happens, we can further count on the information exchanged by the local TTCBs (including the one on the compromised host) through the control channel.

The TTCB security kernel 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 5.2.3. However, we also consider simpler configurations not requiring dedicated trusted hardware for the TTCB security kernel, 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 as long as this construct enjoys the interposition, shielding and validation properties, it is a security kernel. Note that the coverage expected from this configuration cannot be worse than hardened versions of known commercial operating systems, 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 [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 MAFTIA deliverable D24 [Neves & Veríssimo 2001].

5.2.5 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 17. 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 preliminary design of these APIs is explained in more detail in MAFTIA deliverable D24 [Neves & Veríssimo 2001], but this design is expected to evolve as the middleware is developed further.

In Figure 17, 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.

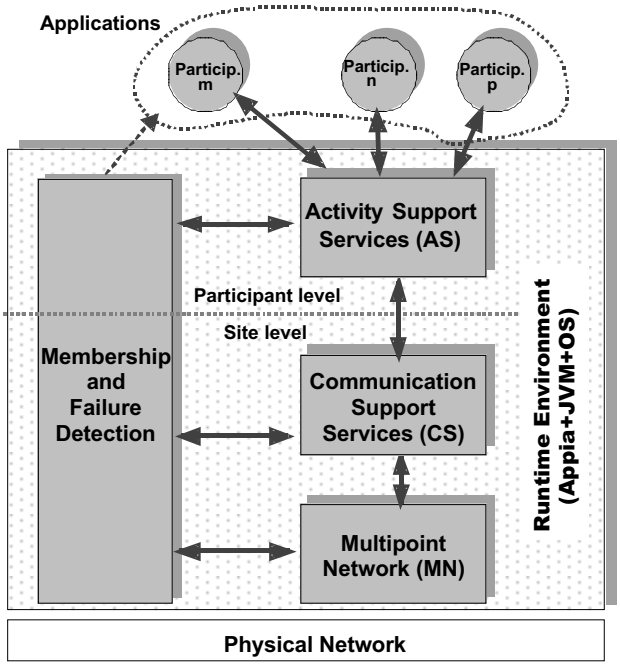


Figure 17 — Detailed architecture of the MAFTIA middleware

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 and intrusion detection.

5.3 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 security kernel (Java Card); fail-controlled with distributed and timed security kernel (TTCB).

The conventions used for the figures in the following sections are as follows: grey means untrusted; 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.

5.3.1.1 Fail-uncontrolled

The fail-uncontrolled or arbitrary failure strategy is based on the no-assumptions attitude discussed in the beginning of the chapter. 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.

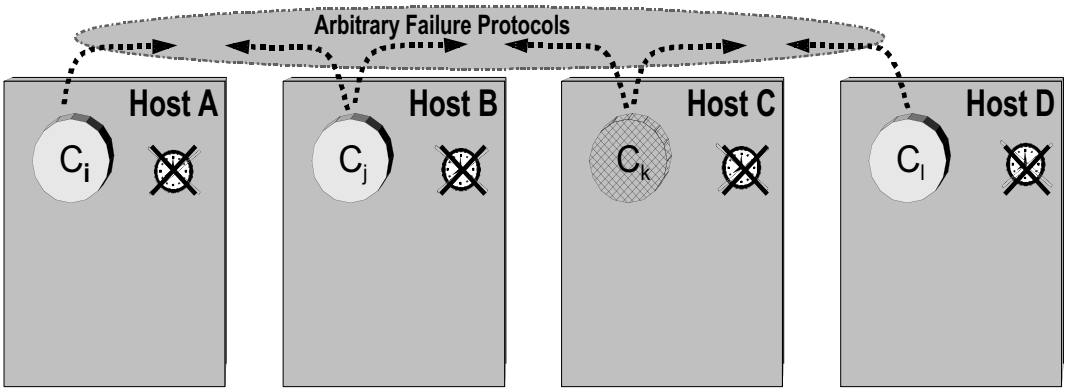


Figure 18 — Fail-uncontrolled

Figure 18 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 C_k 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.

5.3.1.2 Fail-controlled with local security kernel

Figure 19 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 security kernel (LSK), which supports protocols they can trust for certain steps of their operation. This security kernel is implemented in a Java Card that equips the relevant hosts. As such, for a protocol to be able to provide useful service, it has to cope with a mixture of hybrid of arbitrary and fail-silent behaviour, depending on whether a component is interacting with the other components or with the local security kernel.

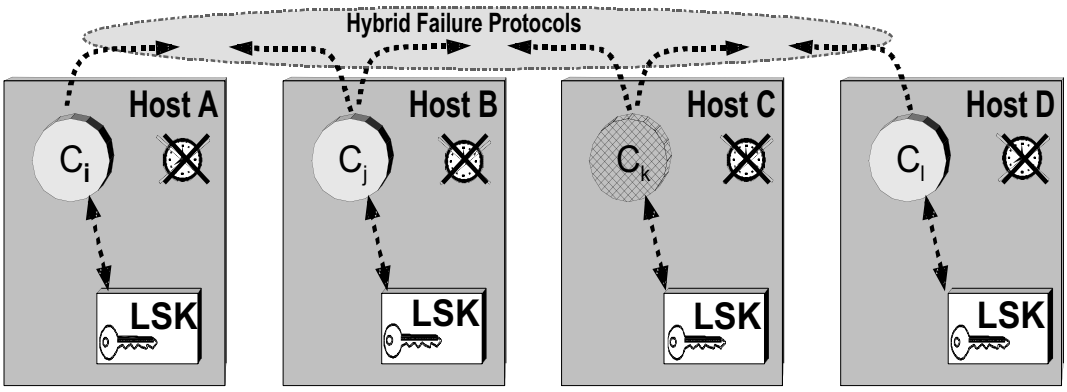


Figure 19 — Fail-controlled with local security kernel

In the example, component C_k 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 security kernel (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 LSK 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 way.

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.

5.3.1.3 Fail-controlled with a TTCB

The "fail-controlled with a TTCB" strategy relies on a distributed and timed security kernel, which serves two purposes: amplifying the degree of trustworthiness of the security kernel support by making it distributed, and supporting timed behaviour in an intrusion resilient way. This strategy consists of assuming, as for the preceding strategies, that the hosts and communication environment are not trusted. However, as suggested by the warped clocks in Figure 20, they are assumed to be partially synchronous.

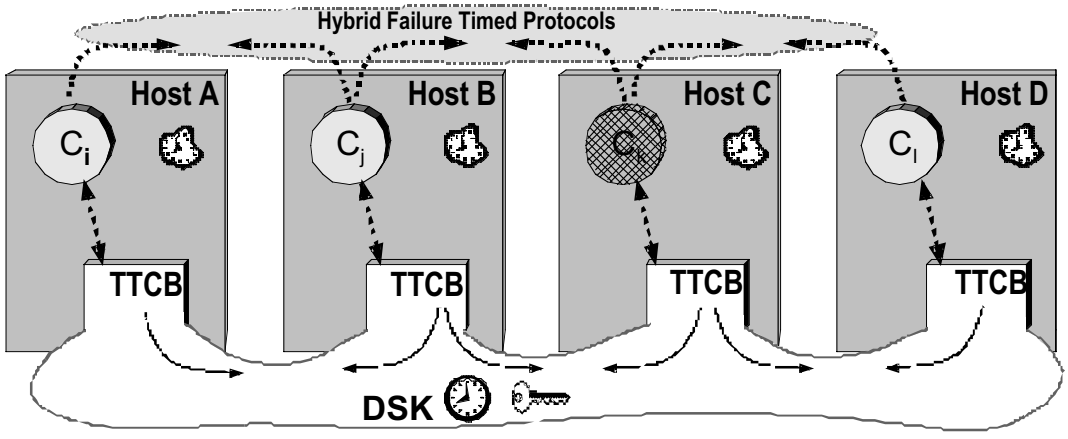


Figure 20 — Fail-controlled with a TTCB

The distributed security kernel (DSK) is implemented by the local TTCBs interconnected by a control network. As with the "fail-controlled with local security kernel" 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 local TTCB. Consider the example of Figure 20, where again component Ck or host C may be arbitrarily malicious. Like the "fail-controlled with local security kernel" strategy, the impact of the faulty behaviour of these components may be limited by enforcing certain validations with the local TTCB. However, with regard to trustworthiness, 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 DSK 'knows' directly what each of the components in different hosts 'say', unlike the solution with LSKs, where an LSK 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 Ci through Cl 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 5.1, 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 20, the TTCB DSK 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.

5.4 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 MAFTIA deliverables D24 [Neves & Veríssimo 2001], D26 [Cachin 2001] and D23 [Abghour *et al.* 2001].

5.4.1 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 5.3, 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 a TTCB" 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 will be explored further in a future deliverable from WP3.

5.4.2 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, Schneider 1990] 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 2001] 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.

MAFTIA deliverable D26 [Cachin 2001] discusses this approach to building dependable trusted third party services in more detail.

5.4.3 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 *security kernel* 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 security kernels on off-the-shelf computers connected to the Internet, critical parts of the security kernel will be implemented on a Java Card.

The implementation of an intrusion-tolerant authorisation service relies on applying error compensation, and the "fail-controlled with local security kernel" 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 security kernel" strategy is used, based on threshold-signature algorithms. Access to application resources is controlled by the local security kernel. If the latter is compromised then the effect of the failure is localised due to the trusted nature of the Java Card: since the Java Card is considered as tamper-proof, the corruption of the local host gives no privilege to access remote objects, and a corrupt host cannot impersonate another host (this would require cloning the Java Card).

For more details of the Authorisation Service, see MAFTIA deliverable D27 [Abghour *et al.* 2001] and a recent publication [Deswarte *et al.* 2001].

5.4.4 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 security kernel” and “fail-controlled with a TTCB”.

Error compensation is implemented using active or “state machine” replication [Powell *et al.* 1988, Schneider 1990]. 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 a TTCB” 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 security kernel” 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 security kernel. In addition, since the security kernel 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 MAFTIA deliverable D24 [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].

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 6.4. In Section 6.3, 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

Chapters 2 and 3 of this deliverable 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, because all mathematical notions of system, components etc. are necessarily abstractions. 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 one would only need new definitions for dependability-specific terms. However, as the scope of MAFTIA includes cryptographic subsystems, this was not possible, because we are not aware 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 MAFTIA deliverable D4 [Adelsbach & Pfitzmann 2001], Chapter 2. 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 an equivalent of Section 2.2: **Atomic systems** (entities, components) are formalised as probabilistic I/O automata[27] 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 5.1.6. We have defined a synchronous and an asynchronous model, both already including the fact that malicious components may want to deviate from timing requirements.[28]

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 Section 6.2.3.

6.2.2 Modelling faults

Due to the mathematical nature of a formalisation, we mainly model errors and failures, not faults ("causes"); thus there is no formalisation of Section 2.3.

In principle, faults may cause a system to turn into an entirely different system. Verifying the dependability of concrete systems, however, clearly presupposes **failure assumptions** as in Section 5.1.1. In other words, we assume that **informal fault forecasting** has preceded the formal modelling. The verification is only meaningful in practice if that forecasting was correct.

The formalisation allows arbitrary failure assumptions by presenting a system as an arbitrary set of possible machine collections. However, it also contains predefined specialisations to several **common failure assumptions.** These include arbitrary or crash failures of individual components, and passive or active tapping of lines. The components can be given a so-called *access structure*, which describes the sets of components that are allowed to fail together.

---

[27] 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.

[28] The asynchronous model may allow the derivation of partially timed models as specializations by introducing specific scheduler components, but this has not been done yet.

Typical access structures are *threshold structures,* where any *t* of *n* components may fail. 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 5.1.1). 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.[29]

6.2.3 Specifications for dependability

As mentioned in Sections 2.1 and 3.1, 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 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.[30]

Readers interested in seeing how **authenticity** and **non-repudiation** can be considered as integrity properties, as claimed in Section 3.1, are referred to the example in Chapter 4 of MAFTIA deliverable D4 [Adelsbach & Pfitzmann 2001].

---

[29] 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.

[30] 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].

6.3 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 and possible in a MAFTIA context.

6.3.1 By security methods

Seven 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 may be somehow formalisable with economic and social models, but we are not considering this here.

   Rigorous models in our sense could, however, support attack prevention by providing precise foundations for risk analysis. This could be achieved by not only specifying and verifying 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.

2. Vulnerability prevention is the process by which one attempts to avoid introducing vulnerabilities during design. Formal, or even semi-formal, specification can help in this process since such specifications are less ambiguous than natural language.

3. Intrusion prevention, beyond the two previous points, has aspects of integrity properties, e.g., the authentication and authorisation systems may be proven. Under the assumption that vulnerabilities remain, however, one cannot derive an overall verification of the desired service from this.

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.

5. 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.

6. Attack forecasting. Like attack prevention, this seems to be accessible to economic and social models rather than rigorous ones in our sense.

7. Vulnerability forecasting. This is accessible to statistical or psychological models rather than rigorous ones, in particular the forecasting of how many vulnerabilities a system will have and how many versions of the system in the field will not have been patched.

6.3.2 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.

6.3.3 By architectural component

An important pre-condition for all component-wise verification is that one can indeed compose systems using only their specifications. Our formalisation of the MAFTIA basic concepts includes the proof of such a composition theorem (see MAFTIA deliverable D4 [Adelsbach & Pfitzmann 2001], Section 2.8). 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. Currently the verification and assessment work-package within MAFTIA employs non-automated proof and automated proof. 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.

6.3.4 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.[31] 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).

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.

6.4 Novel verification work within MAFTIA

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

6.4.1 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. 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. We were indeed able to provide a specification by normal I/O automata that was not hard to translate into the formal language of CSP, used in the work presented in Section 6.4.2, and could easily be translated into other formalisms. The example also led to certain

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

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 proof techniques developed can now be applied to prove further protocols much faster, and several other verifications are under way.

6.4.2 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. It is expected that novel techniques for overcoming this limitation will be required for MAFTIA. However, the model-checking of core MAFTIA concepts has so far required a large amount of careful modelling, as opposed to novel techniques for overcoming size limitations. Our analysis of the synchronous contract signing protocol leads us to believe that there are novel and significant optimisations that could be made, and would be applicable to general deterministic synchronous protocols. Future work will see these ideas integrated with the work on joining cryptography and automated formal analysis, described in Section 6.4.1 above. To date, verifications have been obtained for the asynchronous and synchronous contract-signing protocols. Both protocols have posed no problem to model-check in terms of their size, and were specified in a manner particularly suited to formal analysis. The verification process of the synchronous contract signing protocol was successful in identifying an imprecision that could have led to a security flaw (see Security Method 5, Section 6.3.1 above).

Future work on the verification of MAFTIA concepts and protocols will undoubtedly require the use of more novel approaches to model-checking, such as the use of data-independent and inductive reasoning in constructing proofs for systems of arbitrary size. 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. Structural induction using CSP and FDR [Creese & Reed 1999] facilitates proofs independent of network size, for networks constructed from identical components. Data Independent Induction [Creese 2001] fuses the two methods together allowing us to reason about networks of arbitrary size and of varied topologies, constructed from components that are virtually identical, but vary in their names and knowledge of other components of the network.

The theory of data-independence is constantly expanding to accommodate processes and programs with wider ranges of data-structures. Most recently, results that allow the inclusion of arrays indexed over data-independent types, and storing members of the type, have been developed. This extension will almost certainly be applicable to MAFTIA protocols since it will allow reasoning about nodes in networks whose states can vary with respect to each other over time. One example of this would be the ABBA protocol where nodes can be in different rounds of the protocol at any given time. Work on this is already under way.

Chapter 7 Conclusion

This deliverable contains four main contributions:

- An analysis of attacks, vulnerabilities and intrusions in terms of the basic dependability concepts of fault, error and failure, and the identification of seven 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 TTCB", and "fail-controlled with local security kernel".
- A brief overview of some of the issues concerning the formalisation of the MAFTIA conceptual model, 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 on the MAFTIA middleware (D24), intrusion-detection systems (D3), and the work on validation and assessment (D4). The role of this deliverable (D2) is to describe the basic concepts of dependability and intrusion tolerance that underpin all of the MAFTIA work. These concepts will be developed further as the project continues to refine its ideas about architecture as a result of the experience it gains from prototyping and validating selected components of the overall MAFTIA architecture.

Appendix - Glossary

> This glossary is provided as an aid to reading this document. It should not be considered independently of the body of the document.
>
> For some terms, only dictionary definitions have been given. It was felt necessary to include these terms in the glossary, but the corresponding definitions are particularly subject to change after further scientific discussion.

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 - unintentional.

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 - malicious interaction *fault* aiming to intentionally violate one or more security properties; an *intrusion* attempt.

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 usage [Laprie 1992]; 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 - property of a computer system such that reliance can be justifiably placed on the service it delivers [Laprie 1995].

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* liable to lead to *failure* [Laprie 1992]. 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 fulfilling the *system function,* i.e., from what the system is intended for [Laprie *et al.* 1995]; transition from *correct service* to *incorrect service* [Laprie 1992]; 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 property 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* [Laprie 1992]; *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 ~) - methods and techniques aimed at estimating the present number, the future incidence, and the consequences of *faults*[Laprie *et al.* 1995].

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 ~).*

insurance † - a measure taken to provide for a possible contingency [OMED 1992].

integrity - *dependability* with respect to the non-occurrence of inadequate information alterations [Laprie *et al.* 1995].

intentional - voluntary, deliberate.

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, unleashing devastating consequences for the host system.

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

malicious logic – an internal, intentionally 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 ~) - methods and techniques aimed at preventing *fault* occurrence or introduction [Laprie *et al.* 1995].

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 ~) - methods and techniques aimed at reducing the presence (number, seriousness) of *faults*[Laprie *et al.* 1995].

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* [Laprie 1992].

service (correct ~) - *service* that fulfils the *system function* [Laprie *et al.* 1995].

service (incorrect ~) - *service* that does not fulfil the *system function* [Laprie *et al.* 1995].

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 [Laprie *et al.* 1995].

system user - see *user (system ~)*.

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

tolerance (fault ~) - methods and techniques aimed at providing a *correct service* in spite of *faults* (adapted from [Laprie *et al.* 1995]).

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 - an accidental fault, or a malicious or non-malicious intentional fault, in the requirements, the specification, the design or the configuration of the system, or in the way it is used, 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*, Deliverable D27, MAFTIA Project, January 2001.

[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.

[Akkar *et al.*] M.-L. Akkar, R. Bevan, P. Dischamp and D. Moyart, "Power Analysis, What Is Now Possible...", in *Advances in Cryptology - ASIACRYPT 2000,* Lecture Notes in Computer Science, 1976, pp.489-502, Springer-Verlag.

[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.181-5, 1985.

[Anderson 2001] R. J. Anderson, *Security Engineering,* 640p., John Wiley & Sons, 2001.

[Anderson & Lee 1981] T. A. Anderson and P. A. Lee, *Fault Tolerance — Principles and Practice,* Prentice-Hall, 1981.

[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 1978] A. Avizienis, "Fault Tolerance, the Survival Attribute of Digital Systems", *Proc. of the IEEE*, 66 (10), pp.1109-25, 1978.

[Avizienis *et al.* 2001] A. Avizienis, J.-C. Laprie and B. Randell, *Fundamental Concepts of Dependability*, Research Report N°01145, LAAS-CNRS, April 2001.

[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.

[Blain & Deswarte 1990] L. Blain and Y. Deswarte, "Intrusion-Tolerant Security Server for Delta-4", in *ESPRIT 90 Conference,* (CEC-DG-XIII, Ed.), (Brussels (Belgium)), pp.355-70, Kluwer Academic Publishers, 1990.

[Bracha & Toueg 1985] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols", *Journal of the ACM*, 32 (4), pp.824-40, 1985.

[Cachin 2001] C. Cachin (Ed.), *Specification of Dependable Trusted Third Parties,* Deliverable 26, MAFTIA Project, 2001.

[Cachin *et al.* 2000a] C. Cachin, J. Camenisch, M. Dacier, Y. Deswarte, J. Dobson, D. Horne, K. Kursawe, J.-C. Laprie, J.-C. Lebraud, D. Long, T. McCutcheon, J. Müller, F. Petzold, B. Pfitzmann, D. Powell, B. Randell, M. Schunter, V. Shoup, P. Veríssimo, G. Trouessin, R. J. Stroud, M. Waidner and I. S. Welch, *Reference Model and Use Cases*, Deliverable 1, MAFTIA Project, 2000a.

[Cachin *et al.* 2000b] C. Cachin, K. Kursawe and V. Shoup, "Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography", in *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC),* pp.123-32, 2000b.

[Carter & Schneider 1968] W. C. Carter and P. R. Schneider, "Design of Dynamically Checked Computers", in *IFIP'68 Cong.,* (Amsterdam, The Netherlands), pp.878-83, 1968.

[CEN 13608-1] *Health Informatics - Security for Healthcare Communication - Part 1: Concepts and Terminology*, CEN.

[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, 1996.

[Creese 2001] S. J. Creese, *Data Independent Induction: CSP Model Checking of Arbitrary Sized Networks,* DPhil Thesis, Oxford University, 2001.

[Creese & Reed 1999] S. J. Creese and J. Reed, “Verifying End-To-End Protocols Using Induction with CSP/FDR”, in *4th Int. Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'99),* (Puerto Rico), pp.1243-57, Springer, 1999.

[Cristian 1980] F. Cristian, “Exception Handling and Software Fault Tolerance”, in *10th IEEE Int. Symp. on Fault Tolerant Computing (FTCS-10),* (Kyoto, Japan), pp.97-103, IEEE Computer Society Press, 1980.

[Cristian 1988] F. Cristian, “Agreeing on Who is Present and Who is Absent in a Synchronous Distributed System”, in *18th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-18),* (Tokyo, Japan), pp.206-11, IEEE Computer Society Press, 1988.

[Cristian *et al.* 1985] F. Cristian, H. Aghili, R. Strong and D. Dolev, “Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement”, in *15th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-15),* (Ann Arbor, MI, USA), pp.200-6, IEEE Computer Society Press, 1985.

[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.

[Denning 1982] D. E. Denning, *Cryptography and Data Security,* Addison-Wesley, 1982.

[Deswarte *et al.* 2001] Y. Deswarte, N. Abghour, V. Nicomette and D. Powell, “An internet authorization scheme using smart card-based security kernels”, in *International Conference on Research in Smart Cards (E-smart 2001),* (Cannes, France), Lecture Notes in Computer Science, pp.71-82, Springer-Verlag, 2001.

[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 Computer Society Press, 1991.

[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.

[ECMA TR/46] *Security in Open Systems — A Security Framework*, ECMA.

[Elmendorf 1972] W. R. Elmendorf, “Fault-Tolerant Programming”, in *2nd IEEE Int. Symp. on Fault Tolerant Computing (FTCS-2),* (Newton, MA, USA), pp.79-83, IEEE Computer Society Press, 1972.

[Fabre *et al.* 1994] J.-C. Fabre, Y. Deswarte and B. Randell, “Designing Secure and Reliable Applications using FRS: an Object-Oriented Approach”, in *1st European Dependable Computing Conference (EDCC-1),* (Berlin), Lecture Notes in Computer Science, 852, pp.21-38, Springer-Verlag, 1994.

[Formal Systems (Europe) Ltd] Formal Systems (Europe) Ltd, “Failures-Divergences Refinement”, `http://www.formal.demon.co.uk/FDR2.html`, (accessed: 17 October, 2000).

[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 Computer Society Press, 1986.

[Frison & Wensley 1982] S. G. Frison and J. H. Wensley, “Interactive Consistency and its Impact on the Design of TMR Systems”, in *12th IEEE Int. Symp. on Fault Tolerant Computing (FTCS-12),* (Santa Monica, CA, USA), pp.228-33, IEEE Computer Society Press, 1982.

[Garber 2000] L. Garber, “Denial-of-Service Attacks Rip the Internet”, *Computer*, 33 (4), pp.12-7, 2000.

[Gong 1992] L. Gong, "A Security Risk of Depending on Synchronized Clocks", *Operating Systems Review*, 26 (1), pp.49-53, 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.

[Gray 1978] J. Gray, "Notes on database operating systems", in *Operating Systems: an Advanced Cource. Lecture Notes in Computer Science,* 60, pp.394-481, Springer-Verlag, 1978.

[Gray 1986] J. Gray, "Why do Computers Stop and What can be done about it?", in *5th Symp. on Reliability in Distributed Software and Database Systems,* (Los Angeles, CA, USA), pp.3-12, IEEE Computer Society Press, 1986.

[Halme & Bauer] L. R. Halme and R. K. Bauer, "AINT Misbehaving: A Taxonomy of Anti-Intrusion Techniques", http://www.sans.org/newlook/resources/IDFAQ/aint.htm, (accessed: 10 April, 2000).

[Härder & Reuter 1983] T. Härder and A. Reuter, "Principles of Transaction-Oriented Database Recovery", *Computing Surveys*, 15 (4), pp.287-317, 1983.

[Hoare 1985] C. A. R. Hoare, *Communicating Sequential Processes,* Prentice Hall, 1985.

[Huang & Abraham 1982] K. K. Huang and J. A. Abraham, "Low Cost Schemes for Fault Tolerance in Matrix Operations with Processor Arrays", in *12th IEEE Int. Symp. on Fault Tolerant Computing (FTCS-12),* (Santa Monica, CA, USA), pp.330-7, IEEE Computer Society Press, 1982.

[IETF] IETF, "Intrusion Detection Exchange Format", http://www.ietf.org/html.charters/idwg-charter.html, (accessed: 5 September, 2001).

[ISO 7498-2] *Basic Reference Model, Part 2: Security Architecture*, ISO.

[ITSEC] *Information Technology Security Evaluation Criteria*, Commission of the European Communities.

[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.52-8, IEEE Computer Society Press, 1988.

[Kent & Atkinson 1998] S. Kent and R. Atkinson, *Security Architecture for the Internet Protocol*, Technical Report Request for Comments 2401 IETF, November 1998.

[Kopetz & Ochsenreiter 1987] H. Kopetz and W. Ochsenreiter, "Clock Synchronization in Distributed Real-Time Systems", *IEEE Transactions on Computers*, C-36 (8), pp.933-40, 1987.

[Lala 1986] J. H. Lala, "A Byzantine Resilient Fault-Tolerant Computer for Nuclear Power Plant Applications", in *16th IEEE Int. Symp. on Fault Tolerant Computing (FTCS-16),* (Vienna, Austria), pp.338-43, IEEE Computer Society Press, 1986.

[Lamport & Melliar-Smith 1985] L. Lamport and P. M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults", *Journal of the ACM*, 32 (1), pp.52-78, 1985.

[Lampson 1981] B. W. Lampson, "Atomic Transactions", in *Distributed Systems — Architecture and Implementation,* (B. W. Lampson, Ed.), Lecture Notes in Computer Science, 105, pp.246-65, Springer-Verlag, Berlin, Germany, 1981.

[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, 1994.

[Laprie 1992] J.-C. Laprie (Ed.), *Dependability: Basic Concepts and Terminology,* Dependable Computing and Fault-Tolerance, 5, 265p., Springer-Verlag, Vienna, Austria, 1992.

[Laprie 1995] J.-C. Laprie, "Dependable Computing: Concepts, Limits, Challenges", in *Special Issue, 25th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-25),* (Pasadena, CA, USA), pp.42-54, IEEE Computer Society Press, 1995.

[Laprie *et al.* 1990] J.-C. Laprie, J. Arlat, C. Béounes and K. Kanoun, "Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures", *Computer*, 23 (7), pp.39-51, 1990.

[Laprie *et al.* 1995] J.-C. Laprie, J. Arlat, J.-P. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J.-C. Fabre, H. Guillermain, M. Kaâniche, K. Kanoun, C. Mazet, D. Powell, C. Rabéjac and P. Thévenod, *Dependability Guidebook,* 324p., Cépaduès-Editions, Toulouse, 1995.

[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.

[Lee & Anderson 1990] P. A. Lee and T. Anderson, *Fault Tolerance — Principles and Practice,* Dependable Computing and Fault-Tolerant Systems, 3, Springer-Verlag, Vienna, Austria, 1990.

[LMED 1976] *Longman Modern English Dictionary*, Longman Group Limited, 1976 (O. Watson, Ed.).

[Melliar-Smith & Randell 1977] P. M. Melliar-Smith and B. Randell, "Software Reliability: The Role of Programmed Exception Handling", *ACM SIGPLAN Notices*, 12 (3), pp.95-100, 1977.

[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.

[Nicolaïdis *et al.* 1989] M. Nicolaïdis, S. Noraz and B. Courtois, "A Generalized Theory of Fail-Safe Systems", in *19th IEEE Int. Symp. on Fault Tolerant Computing (FTCS-19),* (Chicago, MI, USA), pp.398-406, IEEE Computer Society Press, 1989.

[Norman 1983] D. A. Norman, "Design Rules Based on Analyses of Human Error", *Communications of the ACM*, 26 (4), pp.254-8, 1983.

[NSA 1998] NSA, "NSA Glossary of Terms Used in Security and Intrusion Detection", National Security Agency, `http://www.sans.org/newlook/resources/glossary.htm`, 1998 (accessed: 5 September, 2001).

[OMED 1992] *The Oxford Modern English Dictionary*, Oxford University Press, 1992 (J. Swannel, Ed.).

[Pease *et al.* 1980] M. Pease, R. Shostak and L. Lamport, "Reaching Agreement in the Presence of Faults", *Journal of the ACM*, 27 (2), pp.228-34, 1980.

[Peterson & Weldon 1972] W. W. Peterson and E. J. Weldon, *Error-Correcting Codes,* MIT Press, 1972.

[Pfitzmann & Waidner 1994] B. Pfitzmann and M. Waidner, *A General Framework for Formal Notions of 'Secure' Systems*, Report N°11/94, Universität Hildesheim, April 1994 (ISSN 0941-3014).

[Porras *et al.*] P. Porras, D. Schnackenberg, S. Staniford-Chen and M. Stillman, "The Common Intrusion Detection Framework Architecture", CIDF working group, `http://www.gidos.org/drafts/architecture.txt`, (accessed: 5 September, 2001).

[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 Computer Society Press, 1988.

[Pradhan 1986] D. K. Pradhan, "Fault-Tolerant Multiprocessor and VLSI-Based System Communication Architectures", in *Fault-Tolerant Computing, Theory and Techniques,* pp.467-576, Prentice Hall, Englewood Cliffs, 1986.

[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, 1989.

[Randell 1975] B. Randell, "System Structure for Software Fault Tolerance", *IEEE Transactions on Software Engineering*, SE-1 (2), pp.220-32, 1975.

[Rennels 1986] D. A. Rennels, "On Implementing Fault-Tolerance in Binary Hypercubes", in *16th IEEE Int. Symp. on Fault Tolerant Computing (FTCS-16),* (Vienna, Austria), pp.344-9, IEEE Computer Society Press, 1986.

[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.130-9, 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.

[Roscoe 1998] A. W. Roscoe, *The Theory and Practice of Concurrency,* Prentice Hall, 1998.

[Schneider 1990] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial", *ACM Computing Surveys*, 22 (4), pp.299-319, 1990.

[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*, Techical Report Request for Comment 1889 Audio-Video Transport Working Group, January 1996.

[Shamir 1979] A. Shamir, "How to Share a Secret", *Communications of the ACM*, 22 (11), pp.612-3, 1979.

[Siewiorek & Johnson 1982] D. P. Siewiorek and D. Johnson, "A Design Methodology for High Reliability Systems: The Intel 432", in *The Theory and Practice of Reliable System Design,* (D. P. Siewiorek and R. S. Swarz, Eds.), pp.621-36, Digital Press, 1982.

[Simmons 1991] G. J. Simmons, "An Introduction to Shared Secret and/or Shared Control Schemes and their Application", in *Contemporary Cryptology: The Science of Information Integrity,* (G. J. Simmons, Ed.), pp.441-97, IEEE Press, 1991.

[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.14-9, IEEE Computer Society Press, 1986.

[Taylor *et al.* 1980] D. J. Taylor, D. E. Morgan and J. P. Black, "Redundancy in Data Structures: Improving Software Fault Tolerance", *IEEE Transactions on Software Engineering*, SE-6 (6), pp.383-94, 1980.

[Trouessin 2000] G. Trouessin, *Towards Trustworthy Security for Healthcare Information Systems*, Report N°GT/2000.03, CESSI/CNAM, June 2000.

[Verissimo *et al.* 2000] P. Verissimo, A. Casimiro and C. Fetzer, "The Timely Computing Base: Timely Actions in the Presence of Uncertain Timeliness", in *Proc. of DSN 2000, the Int. Conf. on Dependable Systems and Networks,* pp.533-52, IEEE/IFIP, 2000.

[Verissimo & Raynal 2000] P. Verissimo and M. Raynal, "Time, Clocks and Temporal Order", in *Recent Advances in Distributed Systems,* (S. Krakowiak and S. K. Shrivastava, Eds.), LNCS vol. 1752, Springer-Verlag, 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, 1997.

[Wakerly 1978] J. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications,* Elsevier North-Holland, New York, 1978.

[Weingart 2000] S. Weingart, "Physical security devices for computer subsystems: A survey of attacks and defenses", in *Workshop on Cryptographic Hardware and Embedded Systems 2000 (CHES 2000),* Lecture Notes in Computer Science, 1965, Springer-Verlag, 2000.

[Yau & Cheung 1975] S. S. Yau and R. C. Cheung, "Design of Self-Checking Software", in *1st. Int. Conf. on Reliable Software,* (Los Angeles, CA, USA), pp.450-7, 1975.

[Ziegler 1976] B. P. Ziegler, *Theory of Modeling and Simulation,* John Wiley, New York, 1976.

[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.