Intrusion-Tolerant Architectures: Concepts and Design *

Paulo Esteves Veríssimo, Nuno Ferreira Neves, Miguel Pupo Correia

Univ. of Lisboa, Faculty of Sciences
Bloco C5, Campo Grande, 1749-016 Lisboa - Portugal
{pjv,nuno,mpc}@di.fc.ul.pt, http://www.navigators.di.fc.ul.pt

Abstract. There is a significant body of research on distributed computing architectures, methodologies and algorithms, both in the fields of fault tolerance and security. Whilst they have taken separate paths until recently, the problems to be solved are of similar nature. In classical dependability, fault tolerance has been the workhorse of many solutions. Classical security-related work has on the other hand privileged, with few exceptions, intrusion prevention. Intrusion tolerance (IT) is a new approach that has slowly emerged during the past decade, and gained impressive momentum recently. Instead of trying to prevent every single intrusion, these are allowed, but tolerated: the system triggers mechanisms that prevent the intrusion from generating a system security failure. The paper describes the fundamental concepts behind IT, tracing their connection with classical fault tolerance and security. We discuss the main strategies and mechanisms for architecting IT systems, and report on recent advances on distributed IT system architectures.

1 Introduction

There is a significant body of research on distributed computing architectures, methodologies and algorithms, both in the fields of dependability and fault tolerance, and in security and information assurance. These are commonly used in a wide spectrum of situations: information infrastructures; commercial web-based sites; embedded systems. Their operation has always been a concern, namely presently, due to the use of COTS, compressed design cycles, openness. Whilst they have taken separate paths until recently, the problems to be solved are of similar nature: keeping systems working correctly, despite the occurrence of mishaps, which we could commonly call faults (accidental or malicious); ensure that, when systems do fail (again, on account of accidental or malicious faults), they do so in a non harmful/catastrophic way. In classical dependability, and mainly in distributed settings, fault tolerance has been the workhorse of the

* Navigators Home Page: http://www.navigators.di.fc.ul.pt. Work partially supported by the EC, through project IST-1999-11583 (MAFTIA), and FCT, through the Large-Scale Informatic Systems Laboratory (LaSIGE), and projects POSI/1999/CHS/33996 (DEFEATS) and POSI/CHS/39815/2001 (COPE).

many solutions published over the years. Classical security-related work has on the other hand privileged, with few exceptions, intrusion prevention, or intrusion detection without systematic forms of processing the intrusion symptoms.

A new approach has slowly emerged during the past decade, and gained impressive momentum recently: intrusion tolerance (IT) [1]. That is, the notion of handling— react, counteract, recover, mask— a wide set of faults encompassing intentional and malicious faults (we may collectively call them intrusions), which may lead to failure of the system security properties if nothing is done to counter their effect on the system state. In short, instead of trying to prevent every single intrusion, these are allowed, but tolerated: the system has the means to trigger mechanisms that prevent the intrusion from generating a system failure.

It is known that distribution and fault tolerance go hand in hand: one distributes to achieve resilience to common mode faults, and/or one embeds fault tolerance in a distributed system to resist the higher fault probabilities coming from distribution. Contrary to some vanishing misconceptions, security and distribution also go hand in hand: one splits and separates information and processing geographically, making life harder to an attacker. This suggests that (distributed) malicious fault tolerance, a.k.a. (distributed) intrusion tolerance is an obvious approach to achieve secure processing. If this is so obvious, why has it not happened earlier?

In fact, the term "intrusion tolerance" has been used for the first time in [19], and a sequel of that work lead to a specific system developed in the DELTA-4 project [16]. In the following years, a number of isolated works, mainly on protocols, took place that can be put under the IT umbrella [10, 31, 22, 2, 24, 4, 21], but only recently did the area develop explosively, with two main projects on both sides of the Atlantic, the OASIS and the MAFTIA projects, doing structured work on concepts, mechanisms and architectures. One main reason is concerned with the fact that distributed systems present fundamental problems in the presence of malicious faults. On the other hand, classical fault tolerance follows a framework that is not completely fit to the universe of intentional and/or malicious faults. These issues will be discussed below.

The purpose of this paper is to make an attempt to systematize these new concepts and design principles. The paper describes the fundamental concepts behind intrusion tolerance (IT), tracing their connection with classical fault tolerance and security, and identifying the main delicate issues emerging in the evolution towards IT. We discuss the main strategies and mechanisms for architecting IT systems, and report on recent advances on distributed IT system architectures. For the sake of clarifying our position, we assume an 'architecture' to be materialized by a given composition of components. Components have given functional and non-functional properties, and an interface where these properties manifest themselves. Components are placed in a given topology of the architecture, and interact through algorithms (in a generic sense), such that global system properties emerge from these interactions.

[1] Example pointers to relevant IT research: MAFTIA: http://www.maftia.org. OASIS: http://www.tolerantsystems.org.

2 The case for Intrusion Tolerance

Dependability has been defined as 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 it is perceptible by its user(s); a user is another system (human or physical) which interacts with the former[5].

Dependability is a body of research that hosts a set of paradigms, amongst which fault tolerance, and it grew under the mental framework of accidental faults, with few exceptions [19, 17], but we will show that the essential concepts can be applied to malicious faults in a coherent manner.

2.1 A brief look at classical fault tolerance and security

Malicious failures make the problem of reliability of a distributed system harder: failures can no longer be considered independent, as with accidental faults, since human attackers are likely to produce "common-mode" symptoms; components may perform collusion through distributed protocols; failures themselves become more severe, since the occurrence of inconsistent outputs, at wrong times, with forged identity or content, can no longer be considered of "low probability"; furthermore, they may occur at specially inconvenient instants or places of the system, driven by an intelligent adversary's mind.

The first question that comes to mind when addressing fault tolerance (FT) under a malicious perspective, is thus: *How do you model the mind of an attacker?*

Traditionally, security has evolved as a combination of: preventing certain attacks from occurring; removing vulnerabilities from initially fragile software; preventing attacks from leading to intrusions. For example, in order to preserve confidentiality, it would be unthinkable to let an intruder read any confidential data at all. Likewise, integrity would assume not letting an intruder modify data at all. That is, with few exceptions, security has long been based on the prevention paradigm. However, let us tentatively imagine the tolerance paradigm in security[1]:

- assuming (and accepting) that systems remain to a certain extent vulnerable;
- assuming (and accepting) that attacks on components/sub-systems can happen and some will be successful;
- ensuring that the overall system nevertheless remains secure and operational.

Then, another question can be put: *How do we let data be read or modified by an intruder, and still ensure confidentiality or integrity?*

2.2 Dependability as a common framework

Let us observe the well-known fault-error-failure sequence in Figure 1. Dependability aims at preventing the failure of the system. This failure has a remote cause, which is a fault (e.g. a bug in a program, a configuration error) which, if activated (e.g. the program execution passes through the faulty line of code),

leads to an error in system state. If nothing is done, failure will manifest itself in system behaviour.

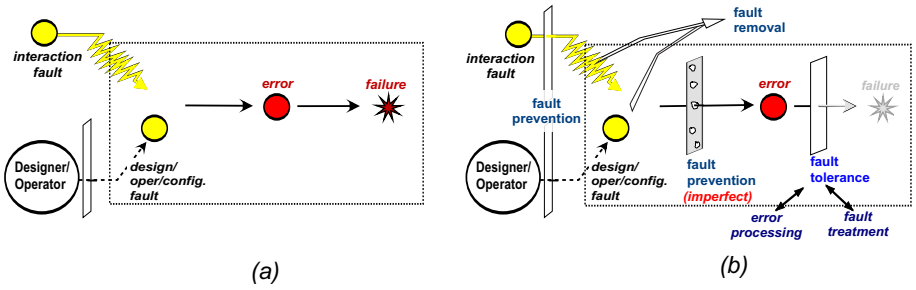

Fig. 1. Fault–> Error–> Failure sequence

In consequence, achieving dependability implies the use of combinations of: fault prevention, or how to prevent the occurrence or introduction of faults; fault removal, or how to reduce the presence (number, severity) of faults; fault forecasting, or how to estimate the presence, creation and consequences of faults; and last but not least, fault tolerance, or how to ensure continued correct service provision despite faults. Thus, achieving dependability vis-a-vis malicious faults (e.g. attacks and vulnerabilities) will mean the combined use of classical prevention and removal techniques with tolerance techniques.

This roadmap seems convincing, but in concrete terms, *how can tolerance be applied in the context of attacks, vulnerabilities, intrusions?*

2.3 Open problems

Let us analyse a few open problems that arise when intrusion tolerance is analysed from a security or fault tolerance background.

To start with, what contributes to the risk of intrusion? Risk is a combined measure of the probability of there being intrusions, and of their severity, that is, of the impact of a failure caused by them. The former is influenced by two factors that act in combination: the level of threat to which a computing or communication system is exposed; and the degree of vulnerability it possesses. The correct measure of how potentially insecure a system can be (in other words, of how hard it will be to make it secure) depends: on the number and nature of the flaws of the system (vulnerabilities); on the potential for there existing attacks on the system (threats). Informally, the probability of an intrusion is given by the probability of there being an attack activating a vulnerability that is sensitive to it. The latter, the impact of failure, is measured by the cost of an intrusion in the system operation, which can be equated in several forms (economical, political, etc.).

Should we try and bring the risk to zero? And is that feasible at all? This is classical prevention/removal: of the number, power, and severity of the vulnerabilities and the attacks the system may be subjected to. The problem is that neither can be made arbitrarily low, for several reasons: it is too costly and/or too complex (e.g., too many lines of code, hardware constraints); certain attacks come from the kind of service being deployed (e.g., public anonymous servers on the Internet); certain vulnerabilities are attached to the design of the system proper (e.g., mechanisms leading to races in certain operating systems).

And even if we could bring the risk to zero, would it be worthwhile? It should be possible to talk about *acceptable risk*: a measure of the probability of failure we are prepared to accept, given the value of the service or data we are trying to protect. This will educate our reasoning when we architect intrusion tolerance, for it establishes criteria for prevention/removal of faults and for the effort that should be put in tolerating the residual faults in the system. Further guidance can be taken for our system assumptions if we think that the hacker or intruder also incurs in a *cost of intruding*. This cost can be measured in terms of time, power, money, or combinations thereof, and clearly contributes to equating 'acceptable risk', by establishing the relation between 'cost of intruding' and 'value of assets'.

How tamper-proof is 'tamper-proof'? Classically, 'tamper-proof' means that a component is shielded, i.e. it cannot be penetrated. Nevertheless, in order to handle the difficulty of finding out that some components were "imperfectly" tamper-proof, experts in the area introduced an alternative designation, 'tamper-resistant', to express that fact. However, the imprecision of the latter is uncomfortable, leading to what we call the "watch-maker syndrome":

- *"Is this watch water-proof?"*
- *"No, it's water-resistant."*
- *"Anyway, I assume that I can swim with it!"*
- *"Well yes, you can! But... I wouldn't trust that very much..."*

A definition is required that attaches a quantifiable notion of "imperfect" to tamper-proofness, without necessarily introducing another vague term.

How can something be trusted and not trustworthy? Classically, in security one aims at building trust between components, but the merits of the object of our trust are not always analysed. This leads to what we called the "unjustified reliance syndrome":

- *"I trust Alice!"*
- *"Well Bob, you shouldn't, she's not trustworthy."*

What is the problem? Bob built trust on Alice through some means that may be correct at a high level (for example, Alice produced some signed credentials). However, Bob is being alerted to a fact he forgot (e.g., that Alice is capable of forging the credentials). It is necessary to establish the difference between what is required of a component, and what the component can give.

How do we model the mind of a hacker? Since the hacker is the perpetrator of attacks on systems, a fault model would be a description of what he/she can

do. Then, a classical attempt at doing it would lead to the "well-behaved hacker syndrome":

- *"Hello, I'll be your hacker today, and here is the list of what I promise not to do."*
- *"Thank you, here are a few additional attacks we would also like you not to attempt."*

In consequence, a malicious-fault modelling methodology is required that refines the kinds of faults that may occur, and one that does not make naive assumptions about how the hacker can act. The crucial questions put in this section will be addressed in the rest of the paper.

3 Intrusion Tolerance concepts

What is Intrusion Tolerance? As said earlier, the tolerance paradigm in security: assumes that systems remain to a certain extent vulnerable; assumes that attacks on components or sub-systems can happen and some will be successful; ensures that the overall system nevertheless remains secure and operational, with a quantifiable probability. In other words:

- faults— malicious and other— occur;
- they generate errors, i.e. component-level security compromises;
- error processing mechanisms make sure that security failure is prevented.

Obviously, a complete approach combines tolerance with prevention, removal, forecasting, after all, the classic dependability fields of action!

3.1 AVI composite fault model

The mechanisms of failure of a system or component, 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). An intrusion has two underlying causes:

Vulnerability - fault in a computing or communication system that can be exploited with malicious intention

Attack - malicious intentional fault attempted at a computing or communication system, with the intent of exploiting a vulnerability in that system

Which then lead to:

Intrusion - a malicious operational fault resulting from a successful attack on a vulnerability

It is important to distinguish between the several kinds of faults susceptible of contributing to a security failure. Figure 2a represents the fundamental sequence of these three kinds of faults: attack → vulnerability → intrusion → failure. This well-defined relationship between attack/vulnerability/intrusion is what we call

the *AVI composite fault model*. The AVI sequence can occur recursively in a coherent chain of events generated by the intruder(s), also called an intrusion campaign. For example, a given vulnerability may have been introduced in the course of an intrusion resulting from a previous successful attack.

Vulnerabilities are the primordial faults existing inside the components, essentially requirements, specification, design or configuration faults (e.g., coding faults allowing program stack overflow, files with root setuid in UNIX, naive passwords, unprotected TCP/IP ports). These are normally accidental, but may be due to intentional actions, as pointed out in the last paragraph. *Attacks* are interaction faults that maliciously attempt to activate one or more of those vulnerabilities (e.g., port scans, email viruses, malicious Java applets or ActiveX controls).

The event of a successful attack activating a vulnerability is called an *intrusion*. This further step towards failure is normally characterized by an erroneous state in the system which may take several forms (e.g., an unauthorized privileged account with telnet access, a system file with undue access permissions to the hacker). Intrusion tolerance means that these errors can for example be unveiled by intrusion detection, and they can be recovered or masked. However, if nothing is done to process the errors resulting from the intrusion, failure of some or several security properties will probably occur.

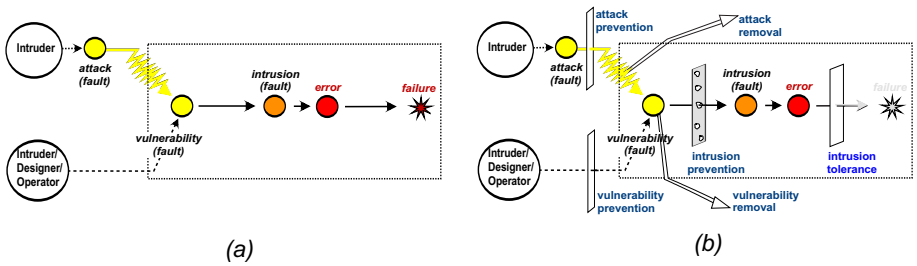

Fig. 2. (a) AVI composite fault model; (b) Preventing security failure

Why a composite model? The AVI model is a specialization of the generic fault → error → failure sequence, which has several virtues. Firstly, it describes the mechanism of intrusion precisely: without matching attacks, a given vulnerability is harmless; without target vulnerabilities, an attacks is irrelevant. Secondly, it provides constructive guidance to build in dependability against malicious faults, through the combined introduction of several techniques. To begin with, we can prevent some attacks from occurring, reducing the level of threat, as shown in Figure 2b. *Attack prevention* can be performed, for example, by shadowing the password file in UNIX, making it unavailable to unauthorized readers, or filtering access to parts of the system (e.g., if a component is behind a firewall and cannot be accessed from the Internet, attack from there is

prevented). We can also perform *attack removal*, which consists of taking measures to discontinue ongoing attacks. However, it is impossible to prevent all attacks, so reducing the level of threat should be combined with reducing the degree of vulnerability, through *vulnerability prevention*, for example by using best-practices in the design and configuration of systems, or through *vulnerability removal* (i.e., debugging, patching, disabling modules, etc.) for example it is not possible to prevent the attack(s) that activate(s) a given vulnerability. The whole of the above-mentioned techniques prefigures what we call *intrusion prevention*, i.e. the attempt to avoid the occurrence of intrusion faults.

Figure 2b suggests, as we discussed earlier, that it is impossible or infeasible to guarantee perfect prevention. The reasons are obvious: it may be not possible to handle all attacks, possibly because not all are known or new ones may appear; it may not be possible to remove or prevent the introduction of new vulnerabilities. For these intrusions still escaping the prevention process, forms of *intrusion tolerance* are required, as shown in the figure, in order to prevent system failure. As will be explained later, these can assume several forms: detection (e.g., of intruded account activity, of trojan horse activity); recovery (e.g., interception and neutralization of intruder activity); or masking (e.g., voting between several components, including a minority of intruded ones).

3.2 Trust and Trustworthiness

The adjectives "trusted" and "trustworthy" are central to many arguments about the dependability of a system. They have been often used inconsistently and up to now, exclusively in a security context[1]. However, the notions of "trust" and "trustworthiness" can be generalized to point to generic properties and not just security; and there is a well-defined relationship between them— in that sense, they relate strongly to the words "dependence" and "dependability".

Trust - the accepted dependence of a component, on a set of properties (functional and/or non-functional) of another component, subsystem or system

In consequence, a trusted component has a set of properties that are relied upon by another component (or components). If A trusts B, then A accepts that a violation in those properties of B might compromise the correct operation of A. Note that trust is not absolute: the degree of trust placed by A on B is expressed by the set of properties, functional and non-functional, which A trusts in B (for example, that a smart card: P1- Gives a correct signature for every input; P2- Has an MTTF of 10h (to a given level of threat...)).

Observe that those properties of B trusted by A might not correspond quantitatively or qualitatively to B's actual properties. However, in order for the relation implied by the definition of trust to be substantiated, trust should be placed *to the extent of* the component's trustworthiness. In other words, trust, the belief that B is dependable, should be placed in the measure of B's dependability.

Trustworthiness - the measure in which a component, subsystem or system, meets a set of properties (functional and/or non-functional)

The trustworthiness of a component is, not surprisingly, defined by how well it secures a set of functional and non-functional properties, deriving from its architecture, construction, and environment, and evaluated as appropriate. A smart card used to implement the example above should actually meet or exceed P1 and P2, in the envisaged operation conditions.

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

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

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

3.3 Coverage and separation of concerns

Let us analyse how to build justified trust under the AVI model. Assume that component C has predicate P that holds with a coverage Pr, and this defines the component's trustworthiness, $\langle P, Pr \rangle$. Another component B should thus trust C to the extent of C possessing P with a probability Pr. So, there can be failures consistent with the limited trustworthiness of C (i.e., that $Pr < 1$): these are "normal", and who/whatever depends on C, like B, should be aware of that fact, and expect it (and maybe take provisions to tolerate the fact in a wider system perspective).

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

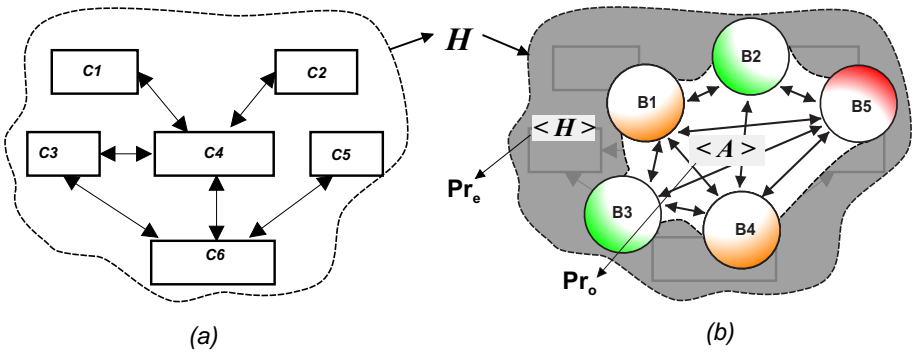

Fig. 3. Building trust

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

Ultimately, what does it mean for component B to trust component C? It means that B assumes something about C. Generalizing, assume a set $\mathcal{B}$ of participants $(B_1 - B_n)$, which run an algorithm offering a set of properties A, on a run-time support environment composed itself of a set $\mathcal{C}$ of components $(C_1 - C_n)$. This modular vision is very adequate for, but not confined to, distributed systems. Imagine the environment as depicted in Figure 3a: $\mathcal{C}$ is architected so as to offer a set of properties, call it H. This serves as the support environment on which $\mathcal{B}$ operates, as suggested by the shaded cushion in Figure 3b.

Observe that $\mathcal{B}$ trusts $\mathcal{C}$ to provide H: $\mathcal{B}$ depends on the environment's properties H to implement the algorithm securing properties A. Likewise, a user of $\mathcal{B}$ trusts the latter to provide A. Without further discourse, this chain of trust would be: if $\mathcal{C}$ is trusted to provide H, then $\mathcal{B}$ is trusted to provide A.

Now let us observe the trustworthiness side. H holds with a probability Pr_e, the environmental assumption coverage[30]:

$Pr_e = Pr(H|f)$, f - any fault

Pr_e measures the trustworthiness of $\mathcal{C}$ (to secure properties H). Given H, A has a certain probability (can be 1 if the algorithm is deterministic and correct, can be less than one if it is probabilistic, and/or if it has design faults) of being fulfilled, the coverage Pr_o or operational assumption coverage:

$Pr_o = Pr(A|H)$

Pr_o measures the confidence on $\mathcal{B}$ securing properties A (given H as environment). Then, the trustworthiness of individual component $\mathcal{B}$ (to secure properties A given H) would be given by Pr_o.

As we propose, these equations should place limits on *the extent* of trust relations. $\mathcal{B}$ should trust $\mathcal{C}$ to the extent of providing H with confidence $Pr_e \leq 1$. However, since the user's trust on $\mathcal{B}$ is implied by $\mathcal{B}$'s trust on $\mathcal{C}$, then the user should trust $\mathcal{B}$ not in isolation, but conditioned to $\mathcal{C}$'s trustworthiness, that is, to the extent of providing A with confidence:

$Pr_a = Pr_o \times Pr_e = Pr(A|H) \times Pr(H|f) = Pr(A|f)$, f - any fault

The resulting chain could go on recursively. Pr_a is the probability that a user of the system composed of $\mathcal{B}$ and $\mathcal{C}$ enjoys properties A, in other words, it measures its trustworthiness.

4 IT frameworks and mechanisms

After introducing intrusion tolerance concepts, we begin this section by briefly analysing the main frameworks with which the architect can work in order to build intrusion tolerant systems: secure and fault-tolerant communication; software-based intrusion tolerance; hardware-based intrusion tolerance; auditing and intrusion detection. We will also look at several known security frameworks[33] under an IT perspective. Then we review error processing mechanisms in order to recover from intrusions.

4.1 Secure and fault-tolerant communication

This is the framework concerning the body of protocols ensuring intrusion tolerant communication. Essentially, relevant to this framework are secure channels and secure envelopes, and classic fault tolerant communication.

Several techniques assist the design of fault-tolerant communication protocols. Their choice depends on the answer to the following question: *What are the classes of failures of communication network components?*

For the architect, this establishes the fundamental link between security and fault tolerance. In classical fault tolerant communication, it is frequent to see omissive fault models (crash, omissions, etc.). In IT the failure mode assumptions should be oriented by the AVI fault model, and by the way specific components' properties may restrict what should be the starting assumption: arbitrary failure (combination of omissive and assertive behaviour). In fact, this is the most adequate baseline model to represent malicious intelligence.

4.2 Software-based intrusion tolerance

Software-based fault tolerance has primarily been aimed at tolerating hardware faults using software techniques. Another important facet is software fault tolerance, aimed at tolerating software design faults by design diversity. Finally, it has long been known that software-based fault tolerance by replication may also be extremely effective at handling transient and intermittent software faults[33].

Let us analyse what can be done under an IT perspective. In the case of design or configuration faults, simple replication would apparently provide little

help: errors would systematically occur in all replicas. This is true from a vulnerability viewpoint: it is bound to exist in all replicas. However, the common-mode syndrome under the AVI model concerns intrusions, or attack-vulnerability pairs, rather than vulnerabilities alone.

This gives the architect some chances. Consider the problem of common-mode vulnerabilities, and of common-mode attacks, i.e. attacks that can be cloned and directed automatically and simultaneously to all (identical) replicas. Design diversity can be applied, for example, by using different operating systems, both to reduce the probability of common-mode vulnerabilities (the classic way), and to reduce the probability of common-mode attacks (by obliging the attacker to master attacks to more than one architecture)[9]. Both reduce the probability of common-mode intrusion, as desired.

However, even mere replication with homogeneous components can yield significant results. How? When components have a high enough trustworthiness that claims can be made about the hardness of achieving a successful attack-vulnerability match on one of them (e.g. "breaking" it). In this case, we could apply the classical principle of achieving a much higher reliability of a replica set than the individual replicas' reliability. For example, simple replication can be used to tolerate attacks, by making it difficult and lengthy for the attacker to launch simultaneous attacks to all replicas with success.

4.3 Hardware-based intrusion tolerance

Software-based and hardware-based fault tolerance are not incompatible design frameworks[33]. In a modular and distributed systems context, hardware fault tolerance today should rather be seen as a means to construct *fail-controlled* components, in other words, components that are prevented from producing certain classes of failures. This contributes to establish improved levels of trustworthiness, and to use the corresponding improved trust to achieve more efficient fault-tolerant systems.

Distributed algorithms that tolerate arbitrary faults are expensive in both resources and time. For efficiency reasons, the use of hardware components with enforced controlled failure modes is often advisable, as a means for providing an infrastructure where protocols resilient to more benign failures can be used, without that implying a degradation in the resilience of the system to malicious faults.

4.4 Auditing and intrusion detection

Logging system actions and events is a good management procedure, and is routinely done in several operating systems. It allows a posteriori diagnosis of problems and their causes, by analysis of the logs. Audit trails are a crucial framework in security.

Intrusion Detection (ID) is a classical framework in security, which has encompassed all kinds of attempts to detect the presence or the likelihood of an

intrusion. ID can be performed in real-time, or off-line. In consequence, an intrusion detection system (IDS) is a supervision system that follows and logs system activity, in order to detect and react (preferably in real-time) against any or all of: attacks (e.g. port scan detection), vulnerabilities (e.g. scanning), and intrusions (e.g. correlation engines).

An aspect deserving mention under an IT viewpoint is the dichotomy between error detection and fault diagnosis, normally concealed in current ID systems[1]. Why does it happen, and why is it important? It happens because IDS are primarily aimed at complementing prevention and triggering manual recovery. It is important because if automatic recovery (fault tolerance) of systems is desired, there is the need to clearly separate: what are errors as per the security policy specification; what are faults, as per the system fault model. Faults (e.g., attacks, vulnerabilities, intrusions) are to be diagnosed, in order that they can be treated (e.g. passivated, removed). Errors are to be detected, in order that they can be automatically processed in real-time (recovered, masked).

ID as error detection will be detailed later in the paper. It addresses detection of erroneous states in a system computation, deriving from malicious action e.g., modified files or messages, OS penetration by buffer overflow. ID as fault diagnosis seeks other purposes, and as such, both activities should not be mixed. Regardless of the error processing mechanism (recovery or masking), administration subsystems have a paramount action w.r.t. fault diagnosis. This facet of classical ID fits into fault treatment[1]. It can serve to give early warning that errors may occur (vulnerability diagnosis, attack forecasting), to assess the degree of success of the intruder in terms of corruption of components and subsystems (intrusion diagnosis), or to find out who/what performed an attack or introduced a vulnerability (attack diagnosis).

4.5 Processing the errors deriving from intrusions

Next we review classes of mechanisms for processing errors deriving from intrusions. Essentially, we discuss the typical error processing mechanisms used in fault tolerance, under an IT perspective: error detection; error recovery; and error masking.

Error detection is concerned with detecting the error after an intrusion is activated. It aims at: confining it to avoid propagation; triggering error recovery mechanisms; triggering fault treatment mechanisms. Examples of typical errors are: forged or inconsistent (Byzantine) messages; modified files or memory variables; phoney OS accounts; sniffers, worms, viruses, in operation.

Error recovery is concerned with recovering from the error once it is detected. It aims at: providing correct service despite the error; recovering from effects of intrusions. Examples of backward recovery are: the system goes back to a previous state known as correct and resumes; the system having suffered DoS (denial of service) attack, re-executes the affected operation; the system having detected corrupted files, pauses, reinstalls them, goes back to last correct point. Forward recovery can also be used: the system proceeds forward to a state that ensures correct provision of service; the system detects intrusion, considers corrupted

operations lost and increases level of security (threshold/quorums increase, key renewal); the system detects intrusion, moves to degraded but safer operational mode.

Error masking is a preferred mechanism when, as often happens, error detection is not reliable or can have large latency. Redundancy is used systematically in order to provide correct service without a noticeable glitch. As examples: systematic voting of operations; Byzantine agreement and interactive consistency; fragmentation-redundancy-scattering; sensor correlation (agreement on imprecise values).

4.6 Intrusion detection mechanisms

As to the methodology employed, classic ID systems belong to one (or a hybrid) of two classes: behaviour-based (or anomaly) detection systems; and knowledge-based (or misuse) detection systems.

Behaviour-based (anomaly) detection systems are characterized by needing no knowledge about specific attacks. They are provided with knowledge about the normal behaviour of the monitored system, acquired e.g., through extensive training of the system in correct operation. As advantages: they do not require a database of attack signatures that needs to be kept up-to-date. As drawbacks: there is a significant potential for false alarms, namely if usage is not very predictable with time; they provide no information (diagnosis) on type of intrusion, they just signal that something unusual happened.

Knowledge-based (misuse) systems rely on a database of previously known attack signatures. Whenever an activity matches a signature, an alarm is generated. As advantages: alarms contain diagnostic information about the cause. The main drawback comes from the potential for omitted or missed alarms, e.g. unknown attacks (incomplete database) or new attacks (on old or new vulnerabilities).

Put under an IT perspective, error detection mechanisms of either class can and should be combined. Combination of ID with automated recovery mechanisms is a research subject in fast progress[1, 14, 23, 11].

5 Intrusion Tolerance strategies

Not surprisingly, intrusion tolerance strategies derive from a confluence of classical fault tolerance and security strategies[33]. Strategies are conditioned by several factors, such as: type of operation, classes of failures (i.e., power of intruder); cost of failure (i.e., limits to the accepted risk); performance; cost; available technology. Technically, besides a few fundamental tradeoffs that should always be made in any design, the grand strategic options for the design of an intrusion-tolerant system develop along a few main lines that we discuss in this section. We describe what we consider to be the main strategic lines that should be considered by the architect of IT systems, in a list that is not exhaustive. Once a strategy is defined, design should progress along the guidelines suggested by the several intrusion-tolerance frameworks just presented.

5.1 Fault Avoidance vs. Fault Tolerance

The first issue we consider is oriented to the system construction, whereas the remaining are related with its operational purpose. It concerns the balance between faults avoided (prevented or removed) and faults tolerated.

On the one hand, this is concerned with the 'zero-vulnerabilities' goal taken in many classical security designs. The Trusted Computing Base paradigm[36], when postulating the existence of a computing nucleus that is impervious to hackers, relies on that assumption. Over the years, it became evident that this was a strategy impossible to follow in generic system design: systems are too complex for the whole design and configuration to be mastered. On the other hand, this balance also concerns attack prevention. Reducing the level of threat improves on the system resilience, by reducing the risk of intrusion. However, for obvious reasons, this is also a very limited solution. As an example, the firewall paranoia of preventing attacks on intranets also leaves many necessary doors (for outside connectivity) closed in its way.

Nevertheless, one should avoid falling in the opposite extreme of the spectrum —assume the worst about system components and attack severity— unless the criticality of the operation justifies a 'minimal assumptions' attitude. This is because arbitrary failure protocols are normally costly in terms of performance and complexity.

The strategic option of using some trusted components— for example in critical parts of the system and its operation— may yield more performant protocols. If taken under a tolerance (rather than prevention) perspective, very high levels of dependability may be achieved. But the condition is that these components be made trustworthy (up to the trust placed on them, as we discussed earlier), that is, that their faulty behaviour is indeed limited to a subset of the possible faults. This is achieved by employing techniques in their construction that lead to the prevention and/or removal of the precluded faults, be them vulnerabilities, attacks, intrusions, or other faults (e.g. omission, timing, etc.).

The recursive (by level of abstraction) and modular (component-based) use of fault tolerance and fault prevention/removal when architecting a system is thus one of the fundamental strategic tradeoffs in solid but effective IT system design. This approach was taken in previous architectural works[29], but has an overwhelming importance in IT, given the nature of faults involved.

5.2 Confidential Operation

When the strategic goal is confidentiality, the system should preferably be architected around error masking, resorting to schemes that despite allowing partial unauthorised reads of pieces of data, do not reveal any useful information. Or schemes that by requiring a quorum above a given threshold to allow access to information, withstand levels of intrusion to the access control mechanism that remain below that threshold. Schemes relying on error detection/recovery are also possible. However, given the specificity of confidentiality (once read, read forever...), they will normally imply some form of forward, rather than backward

recovery, such as rendering the unduly read data irrelevant in the future. They also require low detection latency, to mitigate the risk of error propagation and eventual system failure (in practical terms, the event of information disclosure).

5.3 Perfect Non-stop Operation

When no glitch is acceptable, the system must be architected around error masking, as in classical fault tolerance. Given a set of failure assumptions, enough space redundancy must be supplied to achieve the objective. On the other hand, adequate protocols implementing systematic error masking under the desired fault model must be used (e.g. Byzantine-resilient, TTP-based, etc.). However, note that non-stop availability against general denial-of-service attacks is still an ill-mastered goal in open systems.

5.4 Reconfigurable Operation

Non-stop operation is expensive and as such many services resort to cheaper redundancy management schemes, based on error recovery instead of error masking. These alternative approaches can be characterized by the existence of a visible glitch. The underlying strategy, which we call reconfigurable operation, is normally addressed at availability- or integrity-oriented services, such as transactional databases, web servers, etc.

The strategy is based on intrusion detection. The error symptom triggers a reconfiguration procedure that automatically replaces a failed component by a correct component, or an inadequate or incorrect configuration by an adequate or correct configuration, under the new circumstances (e.g. higher level of threat). For example, if a database replica is attacked and corrupted, it is replaced by a backup. During reconfiguration the service may be temporarily unavailable or suffer some performance degradation, whose duration depends on the recovery mechanisms. If the AVI sequence can be repeated (e.g., while the attack lasts), the service may resort to configurations that degrade QoS in trade for resilience, depending on the policy used (e.g., temporarily disabling a service that contains a vulnerability that cannot be removed, or switching to more resilient but slower protocols).

5.5 Recoverable Operation

Disruption avoidance is not always mandatory, and this may lead to cheaper and simpler systems. Furthermore, in most denial-of-service scenarios in open systems (Internet), it is generically not achievable.

Consider that a component crashes under an attack. An intrusion-tolerant design can still be obtained, if a set of preconditions hold for the component: (a) it takes a lower-bounded time T_c to fall; (b) it takes a upper-bounded time T_r to recover; (c) the duration of blackouts is short enough for the application's needs.

Unlike what happens with classic FT recoverable operation[33], where (c) only depends on (b), here the availability of the system is defined in a more elaborate way, proportionate to the level of threat, in terms of attack severity and duration. Firstly, for a given attack severity, (a) determines system reliability under attack. If an attack lasts less than T_c, the system does not even crash. Secondly, (a) and (b) determine the time for service restoration. For a given attack duration T_a, the system may either recover completely after T_r ($T_a < T_c+T_r$), or else cycle up-down, with a duty cycle of $T_c/(T_c+T_r)$ (longer attacks).

Moreover, the crash, which is provoked maliciously, must not give rise to incorrect computations. This may be achieved through several techniques, amongst which we name secure check-pointing and logging. Recoverable exactly-once operation can be achieved with intrusion-tolerant atomic transactions[33]. In distributed settings, these mechanisms may require secure agreement protocols.

This strategy concerns applications where at the cost of a noticeable temporary service outage, the least amount of redundancy is used. The strategy also serves long-running applications, such as data mining or scientific computations, where availability is not as demanding as in interactive applications, but integrity is of primary concern.

5.6 Fail-Safe

In certain situations, it is necessary to provide for an emergency action to be performed in case the system can no longer tolerate the faults occurring, i.e. it cannot withstand the current level of threat. This is done to prevent the system from evolving to a potentially incorrect situation, suffering or doing unexpected damage. In this case, it is preferable to shut the system down at once, what is called *fail-safe* behaviour. This strategy, often used in safety- and mission-critical systems, is also important in intrusion tolerance, for obvious reasons. It may complement other strategies described above.

6 Modelling malicious faults

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.

What are malicious faults? In the answer to this question lies the crux of the argument with regard to "adequate" intrusion fault models. The term 'malicious' is itself very suggestive, and means a special intent to cause damage. But how do we model the mind and power of the attacker? Indeed, many works have focused on the 'intent', whereas from an IT perspective, one should focus on the 'result'. That is, what should be extracted from the notion of 'maliciousness' is a

technical definition of its objective: the *violation of several or all of the properties of a given service*, attempted in any possible manner within the power available to the intruder.

Classically, failure assumptions fall into essentially two kinds: controlled failure assumptions, and arbitrary failure assumptions.

Controlled failure assumptions specify qualitative and quantitative bounds on component failures. For example, the failure assumptions may specify that components only have timing failures, and that no more than f components fail during an interval of reference. Alternatively, they can admit value failures, but not allow components to spontaneously generate or forge messages, nor impersonate, collude with, or send conflicting information to other components. In the presence of accidental faults this approach is realistic, since it represents very well how common systems work, failing in a benign manner most of the time. However, it can hardly be directly extrapolated to malicious faults, under the above definition of maliciousness.

Arbitrary failure assumptions ideally specify no qualitative or quantitative bounds on component failures. In this context, an arbitrary failure means the capability of generating an interaction at any time, with whatever syntax and semantics (form and meaning), anywhere in the system. Arbitrary failure assumptions adapt perfectly to the notion of maliciousness, but they 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.

Note that the problem lies in how representative are our assumptions vis-a-vis what happens in reality. That is, a problem of *coverage* of our assumptions. So, how to proceed?

6.1 Arbitrary failure assumptions

Consider operations of very high value and/or criticality, such as: financial transactions; contract signing; provision of long term credentials; state secrets. The risk of failure due to violation of assumptions should not be incurred. This justifies considering arbitrary failure assumptions, and building the system around arbitrary-failure resilient building blocks (e.g. Byzantine agreement protocols), despite a possible performance penalty.

In consequence, no assumptions are made on the existence of trusted components such as security kernels or other fail-controlled components. Likewise, a time-free or asynchronous approach must be followed, i.e. no assumptions about timeliness, since timing assumptions are susceptible to be attacked. This limits the classes of applications that can be addressed under these assumptions: 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). So we should seek alternative fault model frameworks to address these requirements under malicious faults.

6.2 Hybrid failure assumptions considered useful

Hybrid assumptions combining several kinds of failure modes would be desirable. There is a body of research, starting with [25] on hybrid failure models that assume different failure type distributions for different nodes. For instance, some nodes are assumed to behave arbitrarily while others are assumed to fail only by crashing. The probabilistic foundation of such distributions might be hard to sustain in the presence of malicious intelligence, unless their behaviour is constrained in some manner. Consider a component or sub-system for which given controlled failure assumptions were made. How can we enforce trustworthiness of the component vis-a-vis the assumed behaviour, that is, coverage of such assumptions, given the unpredictability of attacks and the elusiveness of vulnerabilities?

A composite (AVI) fault model with hybrid failure assumptions is one where the presence and severity of vulnerabilities, attacks and intrusions varies from component to component. Some parts of the system would justifiably exhibit fail-controlled behaviour, whilst the remainder of the system would still be allowed an arbitrary behaviour. This might best be described as *architectural hybridisation*, in the line of works such as [28, 34, 13], where failure assumptions are in fact enforced by the architecture and the construction of the system components, and thus substantiated. That is (*see* Section 3) the component is made *trustworthy* enough to match the *trust* implied by the fail-controlled assumptions.

The task of the architect is made easier since the controlled failure modes of some components vis-a-vis malicious faults restrict the system faults the component can produce. In fact a form of fault prevention was performed at system level: some kinds of system faults are simply not produced. Intrusion-tolerance mechanisms can now be designed using a mixture of arbitrary-failure (fail-uncontrolled or non trusted) and fail-controlled (or trusted) components.

Hybrid failure assumptions can also be the key to secure timed operation. With regard to timeliness and timing failures, hybridisation yields forms of partial synchrony: (i) some subsystems exhibit controlled failure modes and can thus supply timed services in a secure way; (ii) the latter assist the system in fulfilling timeliness specifications; (iii) controlled failure of those specifications is admitted, but timing failure detection can be achieved with the help of trusted components[13].

7 Architecting intrusion-tolerant systems

In this section, we discuss a few notions on architecting intrusion-tolerant systems.

7.1 (Almost) no assumptions

The fail-uncontrolled or arbitrary failure approach to IT architecture is based on assuming as little as possible about the environment's behaviour (faults, synchronism), with the intent of maximizing coverage. It provides a conceptually simple framework for developing and reasoning about the correctness of an algorithm, satisfying safety under any conditions, and providing liveness under certain conditions, normally defined in a probabilistic way.

Randomised Byzantine agreement protocols are an example of typical protocols in this approach. They 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. Of course, for the system as a whole to provide useful service, it is necessary that at least some of the components are correct. This approach is essentially parametric: it will remain correct if a sufficient number of correct participants exist, for any hypothesised number of faulty participants f. Or in other words, with almost no assumptions one is able to achieve extremely resilient protocols.

This has some advantages for the design of secure distributed systems, which is one reason for pursuing such an approach. In fact, sometimes it is necessary and worthwhile to sacrifice performance or timeliness for resilience, for example for very critical operations (key distribution, contract signing, etc.)

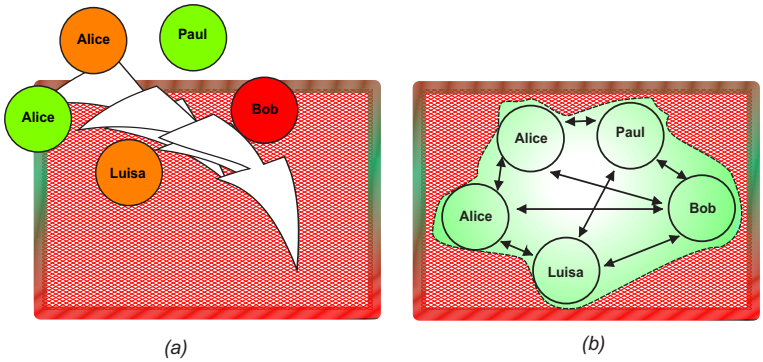

Fig. 4. Arbitrary failure approach

Figure 4 shows the principle in simple terms. The metaphore used from now on is: greyer for hostile, malicious, and whiter for benign, correct. Figure 4a shows the participants being immersed in a hostile and asynchronous environment. The individual hosts and the communication environment are not trusted. Participants may be malicious, and normally the only restriction assumed is in the number of ill-behaved participants. Figure 4b suggests that the protocol, coping with the environment's deficiencies, ensures that the participants collectively provide a correct service (whiter shade).

7.2 Non-justified assumptions, or the power of faith

Alternatively, IT architecture may take the fail-controlled approach. Sometimes, it may simply be assumed that the environment is benign, without substantiating those assumptions. This is often done in accidental fault tolerance, when the environment is reasonably well-known, for example, from statistic measurements. Is it a reasonable approach for malicious faults?

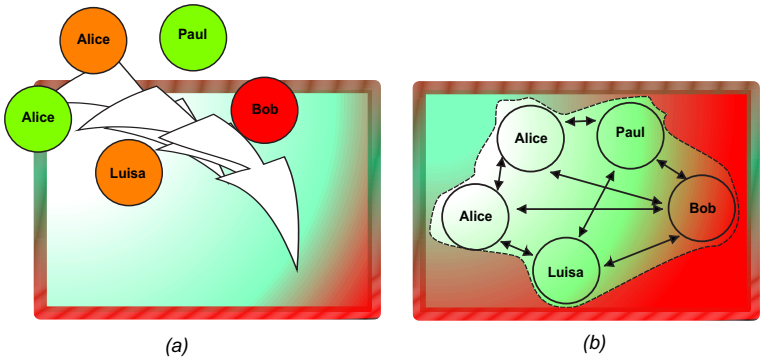

Fig. 5. Non-justified assumptions

Figure 5a shows the participants being immersed in an assumed moderately benign environment (essentially white, with a thin dark part, according to our metaphors). For example, it is usual to consider that the individual hosts (local environment) are trusted, and that the communication environment, though not trusted has a given limited attack model. Some user participants may be malicious.

The implementation is bound to work most of the times. However, it should not be surprising that a behaviour that is assumed out of statistic evidence (or worse, out of faith...) and not by enforcement, can be defrauded by an intruder attacking the run-time environment. Thus, it may turn out that the latter behaves in a manner worse than assumed (e.g., hosts were not that trustworthy, or the communication support was more severely attacked than the model assumed), as suggested in Figure 5b where, say upon an attack, the environment is shown actually more aggressive than initially thought in Figure 5a.

In consequence, making assumptions that are not substantiated in a strong manner may in many cases lead to the lack of trustworthiness (coverage) on the properties of a component or subsystem (suggested in our example by the dark shade partially hitting the participants and protocol). This may be problematic, because it concerns failures not assumed, that is, for which the protocol is not prepared, and which may be orchestrated by malicious intelligence. Their consequences may thus be unpredictable. We discuss a correct approach below.

7.3 Architectural hybridisation

Architectural hybridisation is a solid guiding principle for architecting fail-controlled IT systems. One wishes to avoid the extreme of arbitrary assumptions, without incurring the risks of lack of coverage. Assuming something means trusting, as we saw earlier on, and so architectural hybridisation is an enabler of the approach of *using trusted components*, by making them *trustworthy* enough.

Essentially, the architect tries to make available black boxes with benign behaviour, of omissive or weak fail-silent class[33]. These can have different capabilities (e.g. synchronous or not; local or distributed), and can exist at different levels of abstraction. A good approach is to dress them as run-time environment components, which can be accessed by system calls but provide trustworthy results, in contrast with calls to an untrusted environment. Of course, fail-controlled designs can yield fault-tolerant protocols that are more efficient than truly arbitrary assumptions protocols, but more robust than non-enforced controlled failure protocols.

The tolerance attitude in the design of hybrid IT systems can be characterized by a few aspects:

- assuming as little as possible from the environment or other components;
- making assumptions about well-behaved (trusted) components or parts of the environment whenever strictly necessary;
- enforcing the assumptions on trusted components, by construction;
- unlike classical prevention-based approaches, trusted components do not intervene in all operations, they assist only crucial steps of the execution;
- protocols run thus in an non-trusted environment, single components can be corrupted, faults (intrusions) can occur;
- correct service is built on distributed fault tolerance mechanisms, e.g., agreement and replication amongst participants in several hosts.

7.4 Prevention, Tolerance, and a bit of salt

On achieving trustworthy components, the architect should bear in mind a recipe discussed earlier: the good balance between prevention and tolerance. Let us analyze the principles of operation of a trusted third party (TTP) protocol, as depicted in Figure 6a. Participants Alice, Paul and Bob, run an IT protocol amongst themselves, and trust Trent, the TTP component, to provide a few services that assist the protocol in being intrusion tolerant. What the figure does not show and is seldom asked is: is the TTP trustworthy?

In fact, the TTP is the perfect example of a trusted component that is sometimes (often?) trusted to an extent greater than its trustworthiness.

In Figure 6b we "open the lid" of the TTP and exemplify how a good combination of prevention and tolerance can render it trustworthy. To start with, we require certificate-based authentication, as a means to prevent certain failures from ocurring in the point-to-point interaction of participants with the TTP (e.g., impersonation, forging, etc.). Then, if we replicate the TTP, we make it

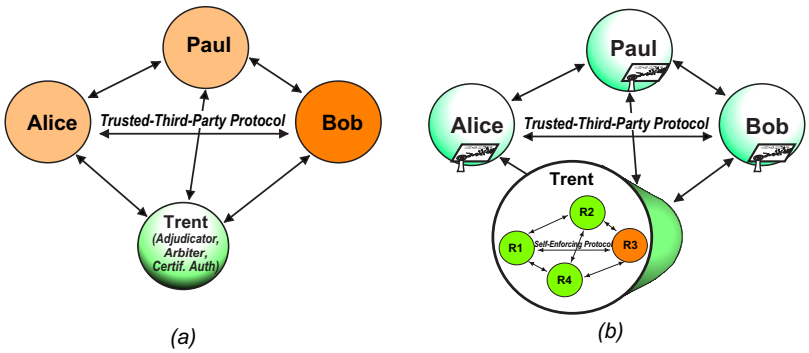

Fig. 6. (a) TTP protocol; (b) Enforcing TTP trustworthiness

resilient to crashes, and to a certain level of attacks on the TTP server replicas, if there is enough redundancy. Furthermore, the replicas should communicate through self-enforcing protocols of the Byzantine-resilient kind, if malicious faults can be attempted at subsets of server replicas.

The user need not be aware of the additional complexity and distribution of the TTP, a usual principle in fault tolerance. In fact, we should "close the lid" so that participants see essentially a single logical entity which they trust (as in Figure 6a). However, by having worked at component level (TTP), we achieve trustworthy behaviour of the component as seen at a higher level (system). Note that in fact, we have prevented some system faults from occurring. This duality prevention/tolerance can be applied recursively in more than one instance. Recently, there has been extensive research on making trustworthy TTPs, for example by recursively using intrusion tolerance mechanisms[1, 38].

7.5 Using trusted components

The relation of trust/trustworthiness can be applied in general when architecting IT systems, as we saw in the last section. However, particular instantiations of trusted components deserve mention here.

IT protocols can combine extremely high efficiency with high resilience if supported by *locally accessible* trusted components. For example, the notion of security kernel in IT would correspond to a fail-controlled local subsystem trusted to execute a few security-related functions correctly, albeit immersed in the remaining environment, subjected to malicious faults.

This can be generalised to any function, such as time-keeping, or failure detection. In that sense, a local trusted component would encapsulate, and supply in a trusted way, a set of functions, considered crucial for protocols and services having to execute in a hostile environment. The use of trusted hardware (e.g. smart cards, appliance boards) may serve to amplify the trustworthiness of these special components. In Figure 7a we see an example of an architecture featuring LTCs (local trusted components). Inter-component communication should

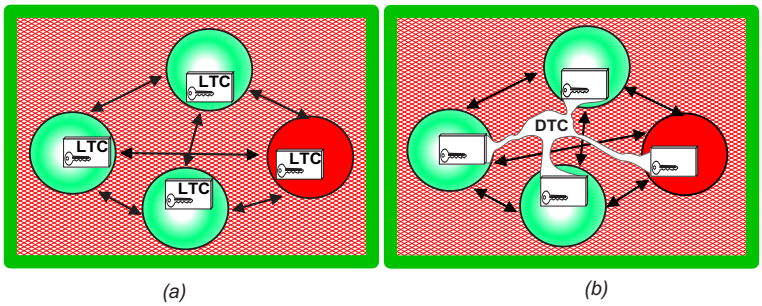

Fig. 7. Using trusted components: (a) Local; (b) Distributed

ensure that correct components enjoy the properties of the LTC despite malicious faults. On the other hand, the implementation of the LTC should ensure that malicious components, such as the one on the right of Figure 7a, do not undermine the operation of the LTC, making it work incorrectly.

Figure 7b shows a distributed trusted component (DTC). It amplifies the power of a LTC, since it assumes the existence of not only local trusted execution, but also a trusted channel among LTCs. This makes it possible to implement distributed trust for low-level operations (e.g., distribution of message authentication codes- MACS). It can be built for example with appliance boards with a private control channel, such as a second network attachment in a host.

A DTC can assist protocols in number of ways, which we discuss with more detail in later sections of the paper, but the fundamental rationale is the following:

- protocol participants have to exchange messages in a world full of threats, some of them may even be malicious and cheat (the normal network);
- there is a channel that correct participants trust, and which they can use to get in touch with each other, even if for rare and short moments;
- they can use this channel to synchronise, disseminate, and agree on, simple but crucial facts of the execution of a protocol, and this limits the potential for Byzantine actions from malicious participants.

8 Some Example Systems

The term “intrusion tolerance” appeared originally in a paper by Fraga and Powell [19]. Later their scheme –Fragmentation-Redundancy-Scattering– was used in the DELTA-4 project to develop an intrusion-tolerant distributed server composed by a set of insecure sites [16].

In the following years a number of isolated IT protocols and systems emerged. BFT [10] is an efficient state-machine replication algorithm [32]. It has been used to implement an intrusion-tolerant NFS server. Rampart provides tools for building IT distributed services: reliable multicast, atomic multicast and membership

protocols [31]. SecureRing is a view-synchronous group communication system based on the Totem single-ring protocols [22]. Both Rampart and SecureRing can be used to build servers using the state-machine replication approach. Fleet [24] use Byzantine quorum systems [2] to build IT data stores, respectively for data abstractions like variables and locks, and for Java objects. The protocol suite CLIQUES supports group key agreement operations for dynamic groups of processes [4, 3]. More recently, two projects have focused on intrusion tolerance, OASIS and MAFTIA, developing several results that will be detailed ahead.

8.1 OASIS

Organically Assured and Survivable Information System (OASIS) [2] is a US DARPA program with the goal of providing "defence capabilities against sophisticated adversaries to allow sustained operation of mission critical functions in the face of known and future cyber attacks against information systems". The program has a strong focus in intrusion tolerance. Its objectives are:

- to construct intrusion-tolerant systems based on potentially vulnerable components;
- to characterize the cost-benefits of intrusion tolerance mechanisms;
- to develop assessment and validation methodologies to evaluate intrusion tolerance mechanisms.

OASIS is financing something like 30 projects. It is not possible to describe all of them so we survey a few that we find interesting and representative.

Intrusion Tolerance by Unpredictable Adaptation (ITUA) aims to develop a middleware to help design applications that tolerate certain classes of attacks [14]. The ITUA architecture is composed by security domains, that abstract the notion of boundaries that are difficult by an attacker to cross (e.g., a LAN protected by a firewall). An intrusion-tolerant application usually has to adapt when there are attacks. ITUA proposes unpredictable adaptation as a means to tolerate attacks that try to predict and take advantage of that adaptation. Adaptation in ITUA is handled by the QuO middleware and group communication is implemented as intrusion-tolerant layers in the Ensemble toolkit.

Intrusion Tolerant Architectures has the objective to develop a methodology based on architectural concepts for constructing intrusion-tolerant systems. The project developed an IT version of Enclaves, a middleware for supporting secure group applications in insecure networks, like the Internet [18]. IT-Enclaves has several leaders from which at most f out of $n \geq 3f+1$ are allowed to be compromised. The leaders provide all group-management services: user authentication, member join and leave, group-key generation, distribution, and refreshment. Each member of the group is in contact with $2f + 1$ leaders.

COCA is an on-line certification-authority for local and wide-area networks [38]. COCA uses replicated servers for availability and intrusion-tolerance. The

[2] http://www.tolerantsystems.org/

certificates that it produces are signed using a threshold cryptography algorithm. COCA assumes an adversary takes a certain time to corrupt a number of servers, therefore from time to time keys are changed (proactive security). Replication is based on a Byzantine quorum system.

8.2 MAFTIA

Malicious- and Accidental-Fault Tolerance for Internet Applications (MAFTIA)[3] is a recently finished EU IST project with the general objective of systematically investigating the 'tolerance paradigm' for constructing large-scale dependable distributed applications. The project had a comprehensive approach that includes both accidental and malicious faults. MAFTIA followed three main lines of action:

- definition of an architectural framework and a conceptual model;
- the design of mechanisms and protocols;
- formal validation and assessment.

The first line aimed to develop a coherent set of concepts for an architecture that could tolerate malicious faults [1]. Work has been done on the definition of a core set of intrusion tolerance concepts, clearly mapped into the classical dependability concepts. The AVI composite fault model presented above was defined in this context. Other relevant work included the definition of synchrony and topological models, the establishment of concepts for intrusion detection and the definition of a MAFTIA node architecture. This architecture includes components such as trusted and untrusted hardware, local and distributed trusted components, operating system and runtime environment, software, etc.

Most MAFTIA work was on the second line, the design of IT mechanisms and protocols. Part of that work was the definition of the *MAFTIA middleware*: architecture and protocols [7]. An asynchronous suite of protocols, including reliable, atomic and causal multicast was defined [8], providing Byzantine resilience by resorting to efficient solutions based on probabilistic execution. Work was also done on protocols based on a timed model, which relies on an innovative concept, the *wormholes*, enhanced subsystems which provide components with a means to obtain a few simple privileged functions and/or channels to other components, with "good" properties otherwise not guaranteed by the "normal" weak environment [35]. For example, the Trusted Timely Computing Base developed in MAFTIA (see next two sections) is based on a wormhole providing timely and secure functions on enviroments that are asynchronous and Byzantine-on-failure. Architectural hybridisation discussed earlier is used to implement the TTCB. In the context of MAFTIA middleware, an IT transaction service with support for multiparty transactions[37] was also designed.

Intrusion detection is assumed as a mechanism for intrusion tolerance but also as a service that has to be made intrusion-tolerant. MAFTIA developed a

[3] http://www.maftia.org/

distributed IT intrusion detection system [15]. Problems like handling high rates of false alarms and combining several IDSs were also explored.

Trusted Third Parties (TTPs) such as certification authorities are important building blocks in today's Internet. MAFTIA designed a generic distributed certification authority that uses threshold cryptography and IT protocols in order to be intrusion-tolerant. Another TTP, the distributed optimistic fair exchange service, was also developed.

MAFTIA defined an *authorization service* based on fine grain protection, i.e., on protection at the level of the object method call [26]. The authorization service is a distributed TTP which can be used to grant or deny authorization for complex operations combining several method calls. The service relies on a local security kernel.

The third line of work was on formalizing the core concepts of MAFTIA and verifying and assessing the work on dependable middleware [27]. A novel rigorous model for the security of reactive systems was developed and protocols were modelled using CSP and FDR.

In the next sections, we describe some of our own work in more detail: the construction of a Trusted Timely Computing Base using the principle of architectural hybridisation, and a protocol using the TTCB wormhole.

Architectural Hybridisation in Practice The Trusted Timely Computing Base (TTCB) is a real-time secure wormhole [13]. The TTCB is a simple component providing a limited set of services. Its architecture is presented in Figure 8. The objective is to support the execution of IT protocols and applications using the architectural hybridisation approach introduced before.

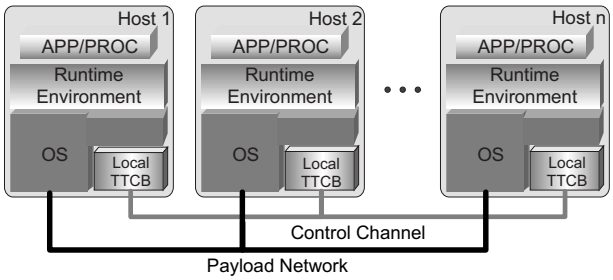

Fig. 8. System architecture with a TTCB

This experimental implementation of the TTCB was based on COTS components. The hosts are common Pentium PCs with a real-time kernel, RT-Linux or RTAI. The hosts are interconnected by two Fast-Ethernet LANs. One corresponds to the payload network in Figure 8, while the other is the TTCB control-channel. It is thus a configuration aimed at local environments, such as sites, campuses, etc. Wide-area configurations are also possible, as discussed in [35].

The design of a system has both functional and non-functional aspects. Next we describe the functionality of the TTCB –its services– and later we discuss the how the security and timeliness (real-time) are enforced in the COTS based TTCB.

The TTCB provides a limited set of services. From the point of view of programming they are a set of functions in a library that can be called by processes in the usual way. We use the word "process" to denominate whatever uses the TTCB services: a normal process, a thread, or another software component.

The TTCB provides three security-related services. The Local Authentication Service allows processes to communicate securely with the TTCB. The service authenticates the local TTCB before a process and establishes a shared symmetric key between both, using a simple authenticated key establishment protocol. This symmetric key is used to secure all their further communication. Every local TTCB has an asymmetric key pair, and we assume that the process manages to get a correct copy of the local TTCB public key. The Trusted Block Agreement Service is the main building block for IT protocols. This service delivers a value obtained from the agreement of values proposed by a set of processes. The service is not intended to replace agreement protocols in the payload system: it works with "small" blocks of data (currently 160 bits), and the TTCB has limited resources to execute it. The service provides a set of functions that can be used to calculate the result. For instance, it can select the value proposed by more processes. A parameter of the service is a timestamp that indicates the last instant when the service starts to be executed. This prevents malicious processes from delaying the service execution indefinitely. The last security-related service is the Random Number Generation Service that provides uniformly distributed random numbers. These numbers can be used as nonces or keys for cryptographic primitives such as authentication protocols.

The TTCB provides also four time services. The Trusted Absolute Timestamping Service provides globally meaningful timestamps. It is possible to obtain timestamps with this characteristic because local TTCBs clocks are synchronized. The Trusted Duration Measurement Service measures the time of the execution of an operation. The Trusted Timing Failure Detection Service checks if a local or distributed operation is executed in an interval of time. The Trusted Timely Execution Service executes special operations securely and within an interval of time inside the TTCB.

RT-Linux and RTAI are two similar real-time engineerings of Linux. Linux was modified so that a real-time executive takes control of the hardware, to enforce real-time behaviour of some real-time tasks. RT tasks were defined as special Linux loadable kernel modules so they run inside the kernel. The scheduler was changed to handle these tasks in a preemptive way and to be configurable to different scheduling disciplines. Linux runs as the lowest priority task and its interruption scheme was changed to be intercepted by RT-Linux/RTAI. The local part of a COTS-based TTCB is basically a (non-real-time) local kernel module, that handles the service calls, and a set of two or more RT tasks that execute all time constrained operations.

The local TTCB is protected by protecting the kernel. From the point of view of security, RT-Linux/RTAI are very similar to Linux. Their main vulnerability is the ability a superuser has to control any resource in the system. This vulnerability is usually reasonably easy to exploit, e.g., using race conditions. Linux capabilities are privileges or access control lists associated with processes that allow a fine grain control on how they use certain objects. However, currently the practical way of using this mechanism is quite basic. There is a system wide *capability bounding set* that bounds the capabilities that can be held by any system process. Removing a capability from that set disables the ability to use an object until the next reboot. Although basic, this mechanism is sufficient to protect the local TTCB. Removing the capability CAP_SYS_MODULE from the capability bounding set we prevent any process from inserting code in the kernel. Removing CAP_SYS_RAWIO we prevent any process from reading and modifying the kernel memory.

For the COTS-based TTCB we make the assumption that the control channel is not accessed physically. Therefore, security has to be guaranteed only in its access points. To be precise, we must prevent an intruder from reading or writing in the control channel access points. This is done by removing the control network device from the kernel so that it can only be accessed by code in the kernel, i.e., by the local TTCB.

The control channel in the COTS-based TTCB is a switched Fast-Ethernet LAN. The timeliness of that network packet is guaranteed preventing packet collisions which would cause unpredictable delays. This requires that: (1) only one host can be connected to each switch port (hubs cannot be used); and (2) the traffic load has to be controlled. The first requirement is obvious. The second is solved by an access control mechanism, that accepts or rejects the execution of a service taking into account the availability of resources (buffers and bandwidth).

A Wormhole-Aware Protocol This section presents an IT protocol based on the TTCB wormhole [4]. This protocol illustrates the approach based on hybrid failure assumptions: most of the system is assumed to fail in an arbitrary way, while the wormhole is assumed to be secure, i.e, to fail only by crashing. The system is also assumed to be asynchronous, except for the TTCB which is synchronous.

The protocol is a *reliable multicast*, a classical problem in distributed systems. Each execution of a multicast has one sender process and several recipient processes. In the rest of the section, we will make the classical separation of *receiving* a message from the network and *delivering* a message – the result of the protocol execution.

A reliable multicast protocol enforces the following two properties [6]: (1) all correct processes deliver the same messages; (2) if a correct sender transmits a message then all correct processes deliver this message. These rules do not imply any guarantees of delivery in case of a malicious sender. However, one of two things will happen, either the correct processes never complete the protocol

[4] The protocol is a simplified version of the protocol presented in [12].

execution and no message is ever delivered, or if they terminate, then they will all deliver the same message. No assumptions are made about the behaviour of malicious (recipient) processes. They might decide to deliver the correct message, a distinct message or no message.

The protocol –BRM (Byzantine Reliable Multicast)– is executed by a set of distributed processes. The processes can fail arbitrarily, e.g., they can crash, delay or not transmit some messages, generate messages inconsistent with the protocol, or collude with other faulty processes with malicious intent. Their communication can also be arbitrarily attacked: messages can be corrupted, removed, introduced, and replayed.

Let us see the process failure modes in more detail. A process is *correct* basically if it follows the protocol until the protocol terminates. Therefore, a process is *failed* if it crashes or deviates from the protocol. There are some additional situations in which we also consider the process to be failed. A process has an identity before the TTCB which is associated to the shared key. If that pair *(id,key)* is captured by an attacker, the process can be impersonated before the TTCB, therefore it has to be considered failed.

Another situation in which we consider a process to be failed is when an attacker manages to disrupt its communication with the other processes. Protocols for asynchronous systems typically assume that messages are repeatedly retransmitted and eventually received (reliable channels). In practice, usually a service which is too delayed is useless. Therefore, BRM retransmits messages a limited number of times and then we assume "isolated" processes to be failed. In channels prone only to accidental faults it is usually considered that no more than Od messages are corrupted/lost in a reference interval of time. Od is the *omission degree* and tests can be made in concrete networks to determine Od with the desired probability. For malicious faults, if a process does not receive a message after $Od + 1$ retransmissions from the sender, with Od computed considering only accidental faults, then it is reasonable to assume that either the process crashed, or an attack is under way. In any case, we will consider the receiver process as failed. The reader, however, should notice that Od is just a parameter of the protocol. If Od is set to a very high value, then BRM will start to behave like the protocols that assume reliable channels.

Formally, a reliable multicast protocol has the properties below [20]. The predicate $sender(M)$ gives the message field with the sender, and $group(M)$ gives the "group" of processes involved, i.e., the sender and the recipients (note that we consider that the sender also delivers).

- *Validity:* If a correct process multicasts a message M, then some correct process in $group(M)$ eventually delivers M.
- *Agreement:* If a correct process delivers a message M, then all correct processes in $group(M)$ eventually deliver M.
- *Integrity:* For any message M, every correct process p delivers M at most once and only if p is in $group(M)$, and if $sender(M)$ is correct then M was previously multicast by $sender(M)$.

BRM-T Sender protocol

1 *tstart = TTCB_getTimestamp() + T_0;*
2 *M := (elist, tstart, data);*
3 *propose := TTCB_propose(elist, tstart, TTCB_TBA_RMULTICAST, H(M));*
4 **repeat** *Od+1 times* **do** *multicast M to elist except sender* **od**
5 *deliver M;*

BRM-T Recipient protocol

6 *read_blocking(M);*
7 *propose := TTCB_propose(M.elist, M.tstart, TTCB_TBA_RMULTICAST, ⊥);*
8 **do** *decide := TTCB_decide(propose.tag);*
9 **while** *(decide.error ≠ TTCB_TBA_ENDED);*
10 **while** *(H(M) ≠ decide.value)* **do** *read_blocking(M)* **od**
11 **repeat** *Od+1 times* **do** *multicast M to elist except sender* **od**
12 *deliver M;*

Fig. 9. BRM protocol.

An implementation of BRM can be found in Figure 9. The sender securely transmits a hash of the message ($H(M)$) to the recipients through the TTCB Agreement Service and then multicasts the message $Od + 1$ times. This hash code is used by the recipients to ensure the integrity and authenticity of the message. When they get a correct copy of the message they multicast it $Od + 1$ times. The pseudo-code is pretty much straightforward so we do not describe it with detail and refer the reader to [12].

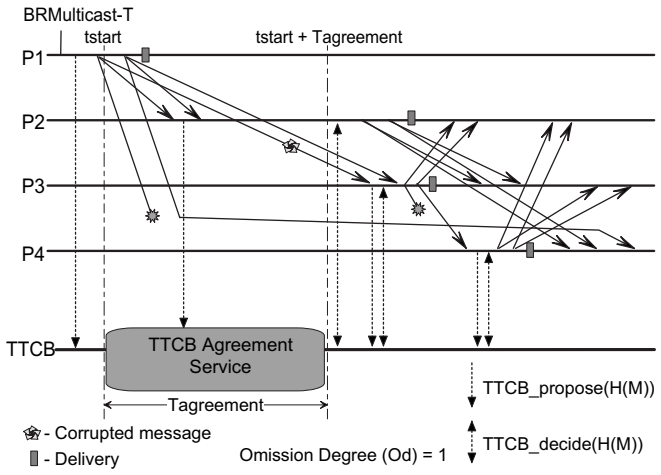

Fig. 10. Protocol execution

Figure 10 illustrates the behavior of the protocol. The horizontal lines represent the execution of processes through time. The thicker line represents the TTCB as a whole, even though, each process calls a separate local TTCB in its host (this representation is used for simplicity). The sender calls the TTCB agreement and then multicasts the message twice ($Od = 1$). These messages are received in the following way: P2 receives the two copies of the message, P3 receives the first copy corrupted and the second well, and P4 does not receive the first copy and the second is delayed. The example assumes that the first message sent to P3 is corrupted only in the *data* part, and for that reason it is still possible to determine this protocol instance. When a message arrives, the recipient calls the TTCB agreement to get the result with the reliable value of $H(M)$. Both processes P2 and P3 get this value almost immediately after the end of the agreement. They use the hash to select which of the messages they received is correct, and then they multicast the message to all the other recipients. P4 asks for the result of the agreement later, when it receives the first message from the protocol. Then, it multicasts the message.

9 Conclusion

We have presented an overview of the main concepts and design principles relevant to intrusion tolerant (IT) architectures. In our opinion, Intrusion Tolerance as a body of knowledge is, and will continue to be for a while, the main catalyst of the evolution of the area of dependability. The challenges put by looking at faults under the perspective of "malicious intelligence" have brought to the agenda hard issues such as uncertainty, adaptivity, incomplete knowledge, interference, and so forth. Under this thrust, researchers have sought replies, sometimes under new names or slight nuances of dependability, such as trustworthiness or survivability.

We believe that fault tolerance will witness an extraordinary evolution, which will have applicability in all fields and not only security-related ones. We will know that we got there when we will no longer talk about accidental faults, attacks or intrusions, but just (and again)... faults.

Acknowledgements

Many of the concepts and design principles presented here derive both from past experience with fault-tolerant and secure system architectures, and from more recent work and challenging discussions within the European IST MAFTIA project. We wish to warmly thank all members of the team, several of whom contributed to IT concepts presented here, and collectively have represented a fenomenal thinking tank.

References

1. Adelsbach, A., Alessandri, D., Cachin, C., Creese, S., Deswarte, Y., Kursawe, K., Laprie, J.C., Powell, D., Randell, B., Riordan, J., Ryan, P., Simmonds, W.,

Stroud, R., Veríssimo, P., Waidner, M., Wespi, A.: Conceptual Model and Architecture of MAFTIA. Project MAFTIA IST-1999-11583 deliverable D21. (2002) http://www.research.ec.org/maftia/deliverables/D21.pdf.
2. Alvisi, L., Malkhi, D., Pierce, E., Reiter, M.K., Wright, R.N.: Dynamic Byzantine quorum systems. In: Proceedings of the IEEE International Conference on Dependable Systems and Networks. (2000) 283–292
3. Amir, Y., Kim, Y., Nita-Rotaru, C., Schultz, J., Stanton, J., Tsudik, G.: Exploring robustness in group key agreement. In: Proceedings of the 21th IEEE International Conference on Distributed Computing Systems. (2001) 399–408
4. Ateniese, G., Steiner, M., Tsudik, G.: New multi-party authentication services and key agreement protocols. IEEE J. of Selected Areas on Communications **18** (2000)
5. Avizienis, A., Laprie, J.C., Randell, B.: Fundamental concepts of dependability. Technical Report 01145, LAAS-CNRS, Toulouse, France (2001)
6. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. Journal of the ACM **32** (1985) 824–840
7. Cachin, C., Correia, M., McCutcheon, T., Neves, N., Pfitzmann, B., Randell, B., Schunter, M., Simmonds, W., Stroud, R., Veríssimo, P., Waidner, M., Welch, I.: Service and Protocol Architecture for the MAFTIA Middleware. Project MAFTIA IST-1999-11583 deliverable D23. (2001) http://www.research.ec.org/maftia/deliverables/D23final.pdf.
8. Cachin, C., Poritz, J.A.: Hydra: Secure replication on the internet. In: Proceedings of the International Conference on Dependable Systems and Networks. (2002)
9. Canetti, R., Gennaro, R., Herzberg, A., Naor, D.: Proactive security: Long-term protection against break-ins. RSA CryptoBytes **3** (1997) 1–8
10. Castro, M., Liskov, B.: Practical Byzantine fault tolerance. In: Proceedings of the Third Symposium on Operating Systems Design and Implementation. (1999)
11. Connelly, K., Chien, A.A.: Breaking the barriers: High performance security for high performance computing. In: Proc. New Security Paradigms Workshop. (2002)
12. Correia, M., Lung, L.C., Neves, N.F., Veríssimo, P.: Efficient Byzantine-resilient reliable multicast on a hybrid failure model. In: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems. (2002) 2–11
13. Correia, M., Veríssimo, P., Neves, N.F.: The design of a COTS real-time distributed security kernel. In: Proceedings of the Fourth European Dependable Computing Conference. (2002) 234–252
14. Cukier, M., Lyons, J., Pandey, P., Ramasamy, H.V., Sanders, W.H., Pal, P., Webber, F., Schantz, R., Loyall, J., Watro, R., Atighetchi, M., Gossett, J.: Intrusion tolerance approaches in ITUA (fast abstract). In: Supplement of the 2001 International Conference on Dependable Systems and Networks. (2001) 64–65
15. Debar, H., Wespi, A.: Aggregation and correlation of intrusion detection alerts. In: 4th Workshop on Recent Advances in Intrusion Detection. Volume 2212 of Lecture Notes in Computer Science. Springer-Verlag (2001) 85–103
16. Deswarte, Y., Blain, L., Fabre, J.C.: Intrusion tolerance in distributed computing systems. In: Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy. (1991) 110–121
17. Dobson, J., Randell, B.: Building reliable secure computing systems out of unreliable insecure components. In: Proceedings of the International Symposium on Security and Privacy, IEEE (1986) 187–193
18. Dutertre, B., Crettaz, V., Stavridou, V.: Intrusion-tolerant Enclaves. In: Proceedings of the IEEE International Symposium on Security and Privacy. (2002)
19. Fraga, J.S., Powell, D.: A fault- and intrusion-tolerant file system. In: Proceedings of the 3rd International Conference on Computer Security. (1985) 203–218

20. Hadzilacos, V., Toueg, S.: A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Department of Computer Science (1994)
21. Hiltunen, M., Schlichting, R., Ugarte, C.A.: Enhancing survivability of security services using redundancy. In: Proceedings of the IEEE International Conference on Dependable Systems and Networks. (2001) 173–182
22. Kihlstrom, K.P., Moser, L.E., Melliar-Smith, P.M.: The SecureRing group communication system. ACM Transactions on Information and System Security **4** (2001) 371–406
23. Knight, J., Heimbigner, D., Wolf, A., Carzaniga, A., Hill, J., Devanbu, P.: The Willow survivability architecture. In: Proceedings of the 4th Information Survivability Workshop. (2001)
24. Malkhi, D., Reiter, M.K., Tulone, D., Ziskind, E.: Persistent objects in the Fleet system. In: Proceedings of the 2nd DARPA Information Survivability Conference and Exposition (DISCEX II). (2001)
25. Meyer, F., Pradhan, D.: Consensus with dual failure modes. In: Proc. of the 17th IEEE International Symposium on Fault-Tolerant Computing. (1987) 214–222
26. Nicomette, V., Deswarte, Y.: An Authorization Scheme for Distributed Object Systems. In: IEEE Symposium on Research in Privacy and Security. (1996) 31–40
27. Pfitzmann, B., Waidner, M.: A model for asynchronous reactive systems and its application to secure message transmission. In: Proceedings of the IEEE Symposium on Research in Security and Privacy. (2001) 184–200
28. Powell, D., Seaton, D., Bonn, G., Veríssimo, P., Waeselynk, F.: The Delta-4 approach to dependability in open distributed computing systems. In: Proceedings of the 18th IEEE International Symposium on Fault-Tolerant Computing. (1988)
29. Powell, D., ed.: Delta-4: A Generic Architecture for Dependable Distributed Processing. Springer-Verlag (1991) Research Reports ESPRIT.
30. Powell, D.: Fault assumptions and assumption coverage. In: Proceedings of the 22nd IEEE International Symposium of Fault-Tolerant Computing. (1992)
31. Reiter, M.K.: The Rampart toolkit for building high-integrity services. In: Theory and Practice in Distributed Systems. Volume 938 of Lecture Notes in Computer Science. Springer-Verlag (1995) 99–110
32. Schneider, F.B.: The state machine approach: A tutorial. Technical Report TR86-800, Cornell University, Computer Science Department (1986)
33. Veríssimo, P., Rodrigues, L.: Distributed Systems for System Architects. Kluwer Academic Publishers (2001)
34. Veríssimo, P., Rodrigues, L., Casimiro, A.: Cesiumspray: a precise and accurate global clock service for large-scale systems. Journal of Real-Time Systems **12** (1997) 243–294
35. Veríssimo, P.: Uncertainty and predictability: Can they be reconciled? In: Future Directions in Distributed Computing. Springer-Verlag LNCS 2584 (2003) –
36. Veríssimo, P., Casimiro, A., Fetzer, C.: The Timely Computing Base: Timely actions in the presence of uncertain timeliness. In: Proceedings of the International Conference on Dependable Systems and Networks. (2000) 533–542
37. Xu, J., Randell, B., Romanovsky, A., Rubira, C., Stroud, R.J., Wu, Z.: Fault tolerance in concurrent object-oriented software through coordinated error recovery. In: Proceedings of the 25th IEEE International Symposium on Fault-Tolerant Computing. (1995) 499–508
38. Zhou, L., Schneider, F., van Renesse, R.: COCA: A secure distributed on-line certification authority. ACM Trans. on Computer Systems **20** (2002) 329–368