Intrusion-Tolerant Middleware: the MAFTIA approach

Paulo E. Veríssimo, Nuno F. Neves,
Christian Cachin, Jonathan Poritz,
David Powell, Yves Deswarte,
Robert Stroud, Ian Welch

DI-FCUL TR–04–14

November 2004

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1700 Lisboa
Portugal

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

Intrusion-Tolerant Middleware: the MAFTIA approach *

Paulo E. Verissimo
Nuno F. Neves
FCUL- U. Lisboa
Lisboa-Portugal
pjv@di.fc.ul.pt
nuno@di.fc.ul.pt

Christian Cachin
Jonathan Poritz
IBM Zurich Research
Rüschlikon-Switzerland
cca@zurich.ibm.com
jap@zurich.ibm.com

David Powell
Yves Deswarte
LAAS-CNRS
Toulouse-France
dpowell@laas.fr
Yves.Deswarte@laas.fr

Robert Stroud
Ian Welch†
U. Newcastle upon Tyne
Newcastle-UK
R.J.Stroud@ncl.ac.uk,
ian.welch@mcs.vuw.ac.nz

Abstract

The pervasive interconnection of systems all over the world has given computer services a significant socio-economic value, which can be affected both by accidental faults and by malicious activity. It would be appealing to address both problems in a seamless manner, through a common approach to security and dependability. This is the proposal of 'intrusion tolerance', where it is assumed that systems remain to some extent faulty and/or vulnerable and subject to attacks that can be successful, the idea being to ensure that the overall system nevertheless remains secure and operational.

In this paper, we report some of the advances made in the European project MAFTIA, namely in what concerns a basis of concepts unifying security and dependability, and a modular and versatile architecture, featuring several intrusion-tolerant middleware building blocks. We describe new architectural constructs and algorithmic strategies, such as: the use of trusted components at several levels of abstraction; new randomization techniques; new replica control and access control algorithms. The paper concludes by exemplifying the construction of intrusion-tolerant applications on the MAFTIA middleware, through a transaction support service.

1 Introduction

The generalized use of computer networks for communication, access to commercial services, research, or simply for entertainment became a reality during the last decade. Some facts emerging from

*This work was partially supported by the EC, through project IST-1999-11583 (MAFTIA), and by the FCT, through LASIGE and projects POSI/1999/CHS/33996 (DEFEATS) and POSI/CHS/39815/2001 (COPE).

†Currently at Victoria U., New Zealand.

this scenario deserve notice: the explosion of the number of users; the explosion of the number of services provided and/or implemented in a distributed way; the pervasive interconnection of systems all over the world.

Some of these services have significant criticality, not only economic (transactions, e-commerce, industry), but also societal (utilities, telecommunications, transportation), demanding well-defined levels of quality of service. However, this objective may be impaired both by accidental faults and by malicious activity: viruses, worms, direct hacker attacks... [1]

The scenario just described is causing a renewed interest in distributed systems security and dependability, which have taken separate paths until recently. The classical approach to Security has mostly consisted in trying to prevent bad things from happening. In other words, the objective has been to try and develop "perfect" systems, systems without vulnerabilities, and/or to detect attacks and intrusions and deploy ad-hoc countermeasures before the system is affected.

It would be appealing to consider an approach where security and dependability would be taken commonly. After all, 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. This is the proposal of *intrusion tolerance*, a new approach that has emerged during the past decade, and gained impressive momentum recently. The idea can be explained very simply [4]:

- to assume and accept that the systems remain to some extent vulnerable;
- to assume and accept that attacks on components can happen and some will be successful;
- to ensure that the overall system nevertheless remains secure and operational.

In this paper, we report some of the advances made in the European project MAFTIA [2]. The path to understanding and conceiving intrusion-tolerant systems starts with revisiting the basic dependability concepts and "reading" them under a security-related perspective, incorporating specific security properties, fault classifications, and security methods. Under the light of these revised concepts, the words "dependence" and "dependability" relate strongly to notions like "trust" and "trustworthiness", giving the latter a powerful and precise meaning: pointing to generic properties and not just security; defining clear relationships between them. These issues are discussed further in Sections 2 and 3.

Surprising as it may seem, intrusion tolerance is not just another instantiation of accidental fault tolerance. Architecting intrusion-tolerant systems, to arrive at some notion of *intrusion-tolerant middleware* for application support, presents multiple challenges, primarily because several of the paradigms and

[1] See, e.g., the CERT Coordination Center statistics at http://www.cert.org/stats/.

[2] MAFTIA portal: http://http://www.newcastle.research.ec.org/maftia.

models used in accidental fault tolerance are not adequate for malicious faults: potential for maliciously caused common-mode faults makes probabilistic assumptions risky (number of "independent" faulty components, fault types); error propagation is the rule rather than the exception (error detection delay, progressive intrusion); typical severity of malicious faults (Byzantine behavior, attacks on timing, contamination of runtime support environment). In addressing these challenges, we devised new architectural constructs and algorithmic strategies, for example, programming models based on the use of trusted components at several levels of abstraction; new randomization and cryptographic techniques; new replica and access control algorithms.

Through the rest of the paper, we start by presenting the rationale behind the main architectural options in MAFTIA, and its blocks and functionality, in Section 3. Section 4 describes different strategies that can be followed in such modular and versatile architectures, to create several instances of the *MAFTIA middleware*. In Sections 5 through Section 7, we digress through instantiations of these strategies in several MAFTIA middleware building blocks, reporting on mechanisms and algorithms researched and prototyped in the project, described with detail in other publications. Finally, in Section 8 we exemplify the construction of intrusion-tolerant applications on the MAFTIA middleware, through a transaction support service that appears to the user as a CORBA-style service, intrusion tolerance being achieved transparently.

2 Basic Concepts

The idea that the *tolerance* approach could be applied to intrusions dates back to the 1980's in some early work on the combination of concepts of dependability and fault-tolerance with security [24, 31, 22]. The sequels of this work [1, 6, 30], which we designate hereafter as the *core dependability concepts*, and contributions from the security and intrusion detection communities [17, 29, 38] have been fundamental in establishing the conceptual and terminological framework that has guided the design process of the MAFTIA architecture. In this section, we outline the main elements of this framework [5].

2.1 Faults, errors and failures

Fundamental to the core dependability concepts is the idea that, at a given level of system abstraction or decomposition, there are three causally related impairments to system dependability that need to be considered. A system *failure* is an event that occurs when the service delivered by the system deviates from correct service. An *error* is that part of the system state that may cause a subsequent failure, whereas a *fault* is the adjudged or hypothesized cause of an error. The notion is recursive in that a failure at one level can be viewed as a fault at the next higher level (e.g., a component failure is a fault seen from the containing system).

The labels given to these concepts conform to standard usage within the dependability community, but the important point we would like to stress is not the words but the fact that there are *three* concepts.

First, it is essential to be able to distinguish the internally observable phenomenon (error) from the externally observable one (failure), which tolerance techniques aim to avert. Indeed, any tolerance technique must be based on some form of detection and recovery acting on internal perturbations before they reach the system's interface to the outside world. The alternative viewpoint, in which any detectable anomaly is deemed to make the system "insecure" in some sense, would make intrusion-*tolerance* an unattainable objective.

Second, the distinction between the internally observable phenomenon (error) and its root cause (fault) is vital since it emphasizes the fact that there may be various plausible causes for the same observed anomaly, including not only an intentionally malicious fault, but also an accidental fault, or an atypical usage profile.

These three notions also have an interesting interpretation in terms of a *security policy*, which we consider as comprising both goals and rules. The *goals* are intended to capture security requirements whose violation would be considered as a *security failure*. Typically, those goals are defined in terms of confidentiality, integrity and availability properties on system services, data or metadata. The *rules* defined in a security policy are lower level constraints on system behavior that aim to ensure that the goals are fulfilled. Violations of the rules do not correspond to security failures but are indicative of errors that could lead to security failures if no precautions are taken.

2.2 Attacks, vulnerabilities and intrusions

We consider an intrusion to be a deliberately malicious software-domain operational[3] fault that has two underlying causes (we refer thus to a *composite* fault model):

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

Vulnerabilities are the primordial faults within the system, in particular design or configuration faults (e.g., coding faults allowing stack overflow, files with root setuid in Unix, naïve passwords, unprotected TCP/IP ports). Vulnerabilities may be introduced during development of the system, or during operation. They may be introduced accidentally or deliberately, with or without malicious intent. As a step in his overall plan of attack, an attacker might introduce vulnerabilities in the form of malicious logic [29].

Attacks may be viewed either at the level of human activity (of the attacker), or at that of the resulting technical activity that is observable within the considered computer system. Attacks (in the technical sense) are malicious faults that attempt to exploit one or more vulnerabilities (e.g., port scans, email viruses, malicious Java applets or ActiveX controls). An attack that successfully exploits a vulnerability results in an intrusion. This further step towards failure is normally characterized by an erroneous state in the system that may take several forms (e.g., an unauthorized privileged account with telnet access, a

[3] As opposed to faults in the hardware domain, e.g., physical sabotage, or to faults introduced during system development, e.g., trapdoors.

system file with undue access permissions for the attacker). Such erroneous states may be corrected or masked by intrusion tolerance but if nothing is done to handle the errors resulting from the intrusion, a security failure will probably occur.

We only qualify the human/technical nature of attacks when necessary; in the absence of qualification, we consider "attack" in its technical sense. *Attacker* is always taken in its human sense, i.e., the malicious person or organization at the origin of attacks. When a technical attack is perpetrated on behalf of the attacker by some piece of code, we refer to the latter as an *attack agent*. Attack agents can be classified according to the following dimensions:

- *dissemination*: propagating (i.e., as in virus or worm); non-propagating (one-off result of an intrusion);
- *trigger conditions*: continuously activated (e.g., an illicit sniffer); serendipitous activation by unsuspecting victim (e.g., Trojan horse); other conditions (specific time, input value, etc.) (i.e., a bomb or a zombie);
- *target of attack*: local (e.g., a bomb or a Trojan horse) or distant (i.e., a zombie);
- *aim of attack*: disclosure (confidentiality); alteration (integrity), denial of service (availability).

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

Finally, when we consider intrusions from an authorization policy viewpoint, we note that they can be subdivided into two types, according to whether an intrusion corresponds to an unauthorized increase in the privilege (set of access rights) of the attacker (or his agent) or to an improper use of authorized operations. We refer to these respectively as *theft* and *abuse* of privilege. Note that theft and abuse of privilege are more general concepts than the often-used notions of "outsider" vs. "insider" intrusions, since (a) a complete "outsider" in an open Internet setting is somewhat difficult to imagine, and (b) an "insider" can attempt both types of intrusions.

2.3 Security methods

The methods underpinning the development of a dependable computing system are classified in the core dependability concepts according to four categories:

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

Fault prevention and fault removal are sometimes grouped together as *fault avoidance*; fault tolerance and fault forecasting constitute *fault acceptance*. Note that avoidance and acceptance should be considered as complementary rather than alternative strategies.

It is enlightening to equate "fault" in these definitions with the notions of attack, vulnerability and intrusion defined above. Taking "attack" in both its human and technical senses leads to ten distinct security-building methods out of a total of sixteen (cf. Table 1).

The focus in this paper is on intrusion-*tolerance* techniques, which should be seen as an additional defense mechanism rather than an alternative to the classic set of techniques grouped under the heading intrusion-*prevention* on Table 1.

2.4 Intrusion-tolerance primitives

In the core dependability concepts, fault-tolerance is defined in terms of error detection and subsequent system recovery, the latter consisting of error handling (aimed at eliminating errors from the system state) and fault handling (aimed at preventing faults from being activated again).

By definition, *error-detection* (and error handling) need to be applied to all errors irrespectively of the specific faults that caused them. However, the *design* of an error-detection technique needs to take into account the hypothesized fault model. For example, detection of errors caused by physical faults requires physical redundancy; detection of those caused by design faults requires diversification redundancy; etc. Detection of errors due to intrusions similarly needs an independent reference to which system activity can be compared. One or more of the following may provide that reference:

- *normal activity profiles*, as in anomaly-detection or behavior-based techniques for intrusion detection [20, 27];
- *undesired activity profiles*, as in misuse-detection or knowledge-based techniques for intrusion detection [20, 27];
- *rules* contained within the system's security policy;
- *activity of peer entities* in trust distribution approaches to intrusion-tolerance such as secret-sharing [48], fragmentation-redundancy-scattering [24], etc.

Error-handling may take three forms:

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

Table 1. Classification of security methods

- *roll-back*: state transformation is carried out by bringing the system back to a previously occupied state, for which a copy (a recovery point, or "checkpoint") has been previously saved — extreme examples include operating system reboots, process re-initialization, and TCP/IP connection re-sets;

- *roll-forward*: state transformation is carried out by finding a new state from which the system can operate — replacement of compromised key shares in threshold-cryptography schemes is an example of this form of error-handling in the context of intrusion tolerance;

- *compensation*: state transformation is carried out by exploiting redundancy in the data representing the erroneous state — masking is the most common form of compensation and is ideally suited for intrusion-tolerance since it can accommodate arbitrarily (e.g., Byzantine) faulty behavior; specific examples include voting, fragmentation-redundancy-scattering and other trust distribution approaches (cf. Sections 5- 8).

Fault handling covers the set of techniques aimed at preventing faults from being re-activated. Whereas error handling is aimed at averting imminent failure, fault handling aims to tackle the underlying causes, whether or not error handling was successful, or even attempted. Three fault-handling primitives can be defined: fault diagnosis, fault isolation and system reconfiguration.

Fault diagnosis is concerned with identifying the type and location of faults that need to be isolated before carrying out system reconfiguration or initiating corrective maintenance. For an intrusion-tolerant system, an essential aspect of diagnosis is the decision as to whether the underlying cause of detected errors was a deliberate attack or an accidental fault. According to the composite fault model presented earlier in this section, fault diagnosis can be further decomposed into:

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

Fault isolation aims to ensure that the source of the detected error(s) is prevented from producing further error(s). In terms of intrusions, this might involve, for example: blocking traffic from components diagnosed as corrupt by changing the settings of firewalls or routers; removing corrupted data from the system; uninstalling software versions with newly-found vulnerabilities; arresting the attacker; etc.

System reconfiguration consists of a redeploying fault-free resources so as to: (a) provide an acceptable, but possibly degraded service while corrective maintenance is carried out on faulty resources, and (b) restore nominal service after corrective maintenance. In an intrusion-tolerant system, possible reconfiguration actions include: software downgrades or upgrades; changing a voting threshold; deployment

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

3 MAFTIA Architecture

The purpose of this section is to introduce the basic models and assumptions underlying the design of the MAFTIA architecture, and then to present an overview of the architecture itself from various perspectives. The section details both the functional aspects of the architecture, and the constructs aimed at achieving intrusion tolerance.

The adjectives "trusted" and "trustworthy" are central to many arguments about the dependability of a system. In the security literature, the terms are often used inconsistently. The MAFTIA notions of "trust" and "trustworthiness" 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* is the reliance put by a component on some properties 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), i.e., there is an *accepted dependence*. The term *trustworthiness* is essentially synonymous to dependability, but is often the preferred term when the focus is on external faults such as attacks.

The definitions above have consequences for the design of intrusion tolerant systems [55] since one can reason separately about trust and trustworthiness. There is separation of concerns between what to do with the trust placed on a component (e.g., designing algorithms that assume that the component exhibits given properties), and how to achieve or show its trustworthiness (e.g., constructing and validating the component that displays the assumed properties). The practical use of these guidelines is exemplified in later sections.

3.1 Models and assumptions

3.1.1 Failure assumptions

A crucial aspect of any fault-tolerant architecture is the fault model upon which the system architecture is conceived, and component interactions are defined. A system fault model is built on assumptions about the way system components fail. Classically, these assumptions fall into two kinds: *controlled failure* assumptions, and *arbitrary failure* assumptions.

Controlled failure assumptions specify constraints on component failures. For example, it may be assumed that components only have timing failures. This approach represents very well how common systems work under the presence of accidental faults, failing in a benign manner most of the time. However, it is difficult to model the behavior of a hacker, so there is a problem of coverage that does not recommend this approach for malicious faults, unless a trustworthy solution can be found.

Arbitrary failure assumptions ideally specify no constraints on component failures. In this context, an arbitrary failure means the capability of generating a message at any time, with whatever syntax and semantics (form and meaning), and sending it to anywhere in the system. Practical systems based on arbitrary failure assumptions do however specify quantitative bounds on the number of failed components. Arbitrary failure assumptions are costly to handle, in terms of performance and complexity, and thus are not compatible with the user requirements of the vast majority of today's on-line applications.

Hybrid failure assumptions combining both kinds of failures are a way out of this dilemma [33]. 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 this behavior is constrained in some manner.

With hybrid assumptions some parts of the system are justifiably assumed to exhibit fail-controlled behavior, whilst the remainder of the system is still allowed an arbitrary behavior. This is an interesting approach for modular and distributed system architectures such as MAFTIA, but one that is only feasible when the fault model is substantiated, that is, the behavior assumed for every single subset of the system can be modeled and/or enforced with high coverage. As a matter of fact, a system normally fails by its weakest link, and naïve assumptions about a component's behavior will be easy prey to hackers.

A first step towards our objective is the organization of the diverse causes of security failure into a *composite fault model* (see Section 2.2), with a well-defined relationship between attack, vulnerability, and intrusion. Such a model allows us to modularize our approach to achieving dependability, by combining different techniques and methods tackling the different classes of faults defined (see Table 1).

3.1.2 Enforcing hybrid failure assumptions

The second step is the enforcement of hybrid failure assumptions. A composite fault model with hybrid failure assumptions is one where the presence and severity of vulnerabilities, attacks and intrusions varies from component to component. Our work might best be described as *architectural hybridization*, in the line of precursor works such as [40] where failure assumptions are in fact enforced by the architecture and the construction of the system components, and thus substantiated.

Consider a component or sub-system for which a given controlled failure assumption is made. How can we achieve coverage of such an assumption, given the unpredictability of attacks and the elusiveness of vulnerabilities? The answer lies in the combined use of intrusion prevention techniques and the implementation of internal intrusion-tolerance mechanisms. The combination of these techniques should be guided by the composite fault model mentioned above (i.e., removing vulnerabilities that are matched by attacks we cannot prevent; preventing or tolerating attacks on vulnerabilities we cannot remove, etc.). In the end, we should justifiably achieve confidence that the component behaves as assumed, failing in a controlled manner, i.e., that the component can be *trusted* because it is *trustworthy*. The measure of this trust is the coverage of the controlled failure assumption.

3.1.3 Intrusion tolerance under hybrid failure assumptions

The approach outlined in the previous sections: establishes a divide-and-conquer strategy for building modular fault-tolerant systems, with regard to failure assumptions; can be applied to achieve different behaviors in different components; can be applied recursively at as many levels of abstraction as are found to be useful. We are now ready to implement our system-level intrusion-tolerance mechanisms, using a mixture of arbitrary-failure and controlled-failure components. By construction, the behavior of the latter vis-à-vis malicious faults is restricted.

As said earlier, in MAFTIA we trust components or subsystems (we will just use the word component henceforth) *to the extent of* their trustworthiness, as perceived at the adequate instances: by the designer, tester, reviewer, user (human or another component), etc. These components can subsequently be used in the construction of fault-tolerant protocols under architectural hybrid failure assumptions.

This is an innovative aspect in MAFTIA that we explore in the following sections. Note that the soundness of the approach does not depend on our making possibly naïve assumptions about what a hacker can or cannot do to a component. In properly designed systems, the trust placed on a component should be qualitatively and/or quantitatively commensurate to its trustworthiness. Likewise, although the accurate provision of such quantification is currently beyond the state-of-the-art, research here is very active, and constitutes one of the most interesting challenges in intrusion-tolerant system architecture and design.

This approach allows us to construct implementations of fault-tolerant protocols that are more efficient than protocol implementations that have to deal with truly arbitrary failure assumptions for all components, and more robust than designs that make controlled failure assumptions without enforcing them.

In our architectural experiments, we devised three main instances of trusted components. The first is based on a Java Card, and is a local component designed to assist the crucial steps of the execution of services and applications. The second is a distributed component (named Trusted Timely Computing Base), based on appliance boards with private network adapters, that is designed to assist crucial steps of the operation of middleware protocols. Whereas these two instances could be best seen as low-level run-time support components, the third instance concerns distributed trusted components in the middleware, recursively built over the low-level trusted components, through distributed fault-tolerance mechanisms.

3.1.4 Arbitrary failure assumptions considered necessary

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

In consequence, an important area of research we pursued is related to arbitrary-failure resilient building blocks, namely communication protocols of the Byzantine class, which do not make assumptions

on the existence of trusted or controlled-failure components. They reason in terms of admitting any behavior from the participants, and allow the corruption of a parameterizable number of participants, say f. The system works correctly as long as there exist $n > 3f$ participants. These protocols do not make assumptions about timeliness either, and are in essence time-free. This has implications on the operational aspects, which will be further discussed in Section 5.

3.2 Architecture description

In this section, we provide an overview of the MAFTIA architecture and discuss the various options that it offers at the hardware, local executive and distributed software levels. The MAFTIA architecture is highly modular. This is an accepted design principle for building distributed fault tolerance into systems. It facilitates the definition of different redundancy strategies for different components, and the placement of the relevant replicas.

3.2.1 Main architectural options

The structure of a MAFTIA host relies on a few main architectural options, some of which are natural consequences of the discussions in the previous section:

The notion of *trusted* — versus untrusted — *hardware*. Most of MAFTIA's hardware is considered to be untrusted, but small parts of it are considered to be trusted to the extent of some quantifiable measure of trustworthiness, for example, being *tamper-proof* by construction. Note that this notion does not necessarily imply proprietary hardware, but for example COTS hardware whose architecture and interface with the rest of the system justifies the aforementioned assumption.

The notion of *trusted support software*. This particular kind of trusted component materializes the notion of a fail-controlled subsystem in the run-time support. It is trusted to execute a few functions correctly (which, given the scope of MAFTIA, will normally be *security-related*) albeit immersed in an environment subjected to malicious faults. The use of trusted hardware may help to substantiate this assumption.

The notion of *run-time environment*, extending operating system capabilities and hiding heterogeneity amongst host operating systems by offering a homogeneous API and framework for protocol composition. Functions supplied by the above-mentioned trusted support software are offered through the run-time API.

Modular and multi-layered *middleware*, with a neat separation between: the multipoint network abstraction, the communication support services, and the activity support services. A given middleware layer may implement another instantiation of a trusted MAFTIA component: a *trusted distributed component* that overcomes the faulty behavior of lower layers and provides certain functions in a trustworthy way, characterized by a given failure semantics (resilience in number and severity of faults).

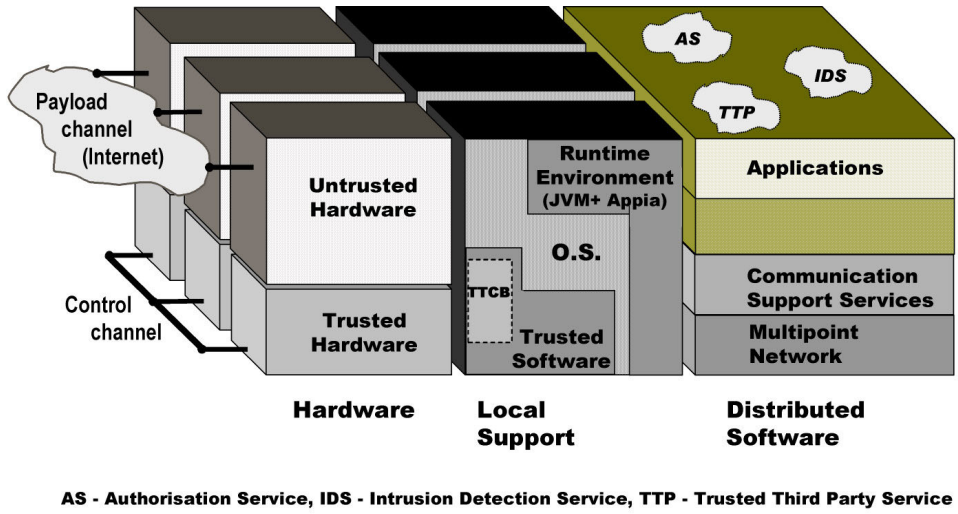

Figure 1. MAFTIA architecture dimensions

The MAFTIA architecture can be depicted in at least three different dimensions (*see* Figure 1). First, there is the *hardware* dimension, which includes the host and networking devices that make up the physical distributed system. Second, within each node, there are the *local support* services provided by the operating system and the run-time platform. These may vary from host to host in a heterogeneous system, and some services may even not be available on some hosts or may have to be accessed via the network using protocols providing an appropriate degree of trust. However, at a minimum, the local services include typical operating system functionality such as the ability to run processes, send messages across the network, access local persistent storage (if it exists), etc. Third, there is the *distributed software* provided by MAFTIA: the layers of *middleware*, running on top of the run-time support mechanisms provided by each host; and MAFTIA's native *services*, depicted in the picture — authorization, intrusion detection, and trusted third party services. Applications built to run on top of MAFTIA use the abstractions provided by the middleware and the application services to operate securely across several hosts, and/or be accessed securely by users running on remote nodes, even in the presence of malicious faults.

3.2.2 Hardware

We assume that the hardware in individual MAFTIA hosts is untrusted in general. Most of a host's operations run on untrusted hardware, e.g., the usual machinery of a PC or workstation, connected through the normal networking infrastructure to the Internet, which we call the *payload channel*. However, some hosts (*see* Figure 1) may have pieces of hardware that are trusted to the extent of being regarded as tamper-proof, i.e., we assume that intruders do not have direct access to the inside of the component.

Some hosts, for example, servers, will have trusted hardware components. Currently, we consider two

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

Note that, contrary to the traditional security view of the term "tamper-resistance" to denote a downgraded version of "tamper-proof-ness", we separate concerns between what is assumed ("tamper- proofness") and the merit of that assumption (its coverage), which may be imperfect. For example, the Java Card is assumed in MAFTIA terminology to be tamper-proof, but this quality is trusted to the extent we believe it is worthy of that trust. The next section shows that trust to be a limited one.

3.2.3 Local support

The local support dimension of the architecture (*see* Figure 1) consists essentially of the operating system augmented with appropriate extensions. We have adopted Java as a platform-independent and object-oriented programming environment, and thus our middleware, service and application software modules are constructed to run on the Java Virtual Machine (JVM) run-time environment. The MAFTIA run-time support also includes the APPIA protocol kernel [34] which supports the construction of middleware protocols from the composition of micro-protocols. The run-time support thus includes abstractions of typical local platform services such as process execution, inter-process communication, access to local persistent storage, and protocol management, enhanced with specialized functions provided by the trusted support software, implemented in two components, the Java Card Module (JCM) and the Trusted Timely Computing Base (TTCB).

The Java Card Module (JCM) is used to assist the operation of a reference monitor, which supports the MAFTIA Authorization Service (*see* Section 7). The reference monitor checks all accesses to local objects, whether persistent or transient, and autonomously manages all access rights for local transient objects. The JCM runs partly on the operating system kernel (the reader interface part) and partly on the Java Card (the function's logic and the data structures, e.g., keys). Software components interact with it through the run-time support (the JVM). The Java Card is trusted to the following extent: the effort, in means or time, necessary to subvert it is incommensurate with the consequences of violating the assumption of JCM resilience.

The Trusted Timely Computing Base (TTCB) is a distributed trusted support component responsible for providing a basic set of trusted services related to time and security, to middleware protocols (communication and activity support). The TTCB is designed to act as an assistant for parts of the execution

of the protocols and applications supported by the MAFTIA middleware, and consequently it can be called from any level of the middleware dimension of the architecture. It aims to support malicious-fault tolerant protocols of any synchrony built to a fail-controlled model, such as reliable multicast, by supplying reliable failure-detection and other control information dissemination. In essence, this component implements some degree of distributed trust for low-level operations. That is, protocol participants essentially exchange their messages in a world full of threats, some of them may even be malicious and cheat, but there is an oracle that correct participants can trust, and a channel that they can use to get in touch with each other, even for rare moments. Moreover, this oracle also acts a point of synchronization for all participants, which limits the potential for Byzantine action (inconsistent value faults) by malicious protocol participants. The other important characteristic is that the TTCB is synchronous, in the sense of having reliable clocks and being able to execute timely functions. Furthermore, the control channel provides timely (synchronous) and ordered communication among TTCB modules, providing simple ways to work around the FLP impossibility result. A local TTCB runs partly on the operating system kernel (the appliance board interface part), and partly on the appliance board itself. Software components interact with it through the run-time support (the JVM). The TTCB component is trusted to the following extent: it is assumed to be not feasible to subvert the TTCB, but it may be possible to interfere in its interaction with software components through the JVM. Whilst we let a local host be compromised, we make sure that it does not undermine the distributed TTCB operation.

The TTCB would normally be built on dedicated hardware modules, with a dedicated network. However, we have also designed simpler configurations not requiring dedicated trusted hardware for the TTCB. The software-based solution consists of a small secure real-time kernel running on the bare machine hardware, inside which the TTCB is built, and on top of which the regular operating system runs (and all the rest of the host software) [19]. Note that the coverage expected of this configuration cannot be worse than security-hardened versions of known commercial operating systems. It might actually be better, since it only addresses the inner kernel and not the operating system as a whole. It may thus constitute a very attractive implementation principle, for MAFTIA and in general, for its cost/simplicity/resilience trade-off. The control channel can also assume several forms exhibiting different levels of timeliness and resilience, as detailed in [19]: it may or may not be based on a physically different network from the one supporting the *payload* channel; secure virtual private networks linking all TTCB modules together can be built over alternative networks, such as ISDN or GSM/UMTS.

3.2.4 Middleware

The distribution dimension impacts on the protocol design but not on the services provided by each host. These are constructed on the functionality provided by the several middleware modules, represented in Figure 1. These interactions occur through the run-time environment. The several profiles for building protocols, which will be detailed in the sections ahead, are achieved by composition of

the micro-protocols necessary to achieve the desired quality of service. The middleware hides these distinctions from the application programmer by providing uniform APIs that are parameterized with functional and non-functional guarantees. The design of these APIs is explained in more detail in [35].

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

Figure 2 details the middleware layers. We distinguish between site and participant parts, depending on whether the functionality provided is host-global or not, respectively. The site part has access to and depends on a physical networking infrastructure, not represented for simplicity, and multiplexes host-global services to any participant-level module. The participant part offers support to local participants (e.g., user applications) engaging in distributed computations. The lowest layer is the *Multipoint Network* module, *MN*, created over the physical infrastructure. Its main properties are the provision of multipoint addressing, basic secure channels, and management communications. The MN layer hides the particularities of the underlying network to which a given site is directly attached, and is as thin as the intrinsic properties of the former allow. It also provides a run-time (JVM and APPIA) compliant interface for the protocols to be used (e.g., IP, IPSEC, SNMP).

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

The block on the left of the figure implements failure detection and membership management. *Site failure detection* is in charge of assessing the connectivity and correctness of sites, whereas *participant failure detection* assesses the liveness of local participants, based on local information provided by sensors in the operating system and run-time support. *Membership* management, which depends on failure information, creates and modifies the membership (registered members) and the view (currently active, or non-failed, or trusted members), of sets of sites and of participant groups. Both the AS and CS modules depend on this information.

4 Intrusion Tolerance Strategies in MAFTIA

The goal of MAFTIA is to support the construction of dependable trustworthy applications, implemented by collections of components with varying degrees of trustworthiness. This is achieved by relying on distributed fault and intrusion-tolerance mechanisms. Given the variety of possible MAFTIA ap-

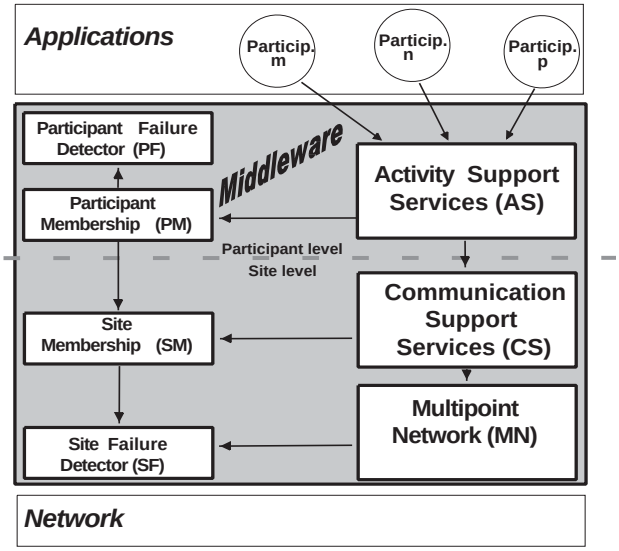

Figure 2. Detail of the MAFTIA middleware

plications, several different architectural strategies are pursued in order to achieve the above-mentioned goal. These strategies are applied at several levels of abstraction of the architecture, most importantly, in the implementation of the middleware and application services. In this section, we describe these strategies: fail-uncontrolled or arbitrary; fail-controlled with local trusted components; fail-controlled with distributed trusted components.

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

4.0.5 Fail-uncontrolled

The fail-uncontrolled or arbitrary failure strategy is based on the no-assumptions attitude discussed in Section 3. When very large coverage is sought of given mechanisms in MAFTIA, we resort to making no assumptions about time, following an asynchronous model, and we make essentially no assumptions about the faulty behavior of either the components or the environment. Of course, for the system as a whole to provide useful service, it is necessary that at least some of the components are correct. This approach is essentially parametric: it will remain correct if a sufficient number of correct participants exist, for any hypothesized number of faulty participants f.

Figure 3 shows the principle in simple terms. The hosts and the communication environment are not trusted, and are fully asynchronous. For a protocol to be able to provide correct service, it must cope

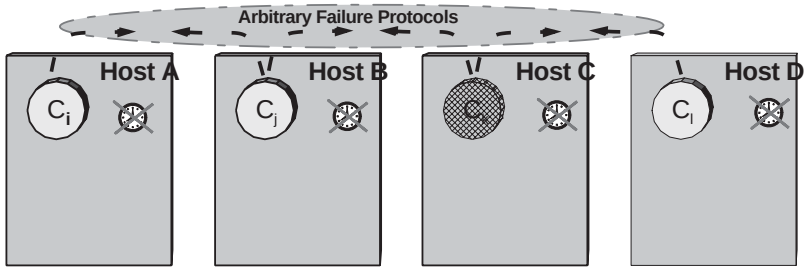

Figure 3. Fail-uncontrolled

with arbitrary failures of components and the environment. For example, component Ck is malicious, but this may be because the component itself or host C have been tampered with, or because an intruder in the communication system simulates that behavior.

Some protocols used by the MAFTIA middleware follow this strategy, in order to be resilient to arbitrary failure assumptions. They are of the probabilistic Byzantine class, and require a number of hosts $n > 3f$, for f faulty components. The MAFTIA middleware provides different qualities of service in this asynchronous profile (see Section 5), achieved by composition of several micro-protocols on top of basic binary Byzantine agreement, in order to achieve: reliable broadcast, atomic broadcast; multi-valued Byzantine agreement.

4.0.6 Fail-controlled with local trusted components

Figure 4 exemplifies a fail-controlled strategy. It consists of assuming that, as for the fail-uncontrolled strategy, hosts and communication environment are not trusted, and asynchronous. However, hosts have a local trusted component (LTC), which supports functions they can trust for certain steps of their operation. In MAFTIA, this strategy is implemented through a Java Card that equips some hosts. As such, we can construct protocols that cope with a hybrid of arbitrary and fail-silent behavior, depending on whether a component is interacting with the other components or with the local trusted component (LTC).

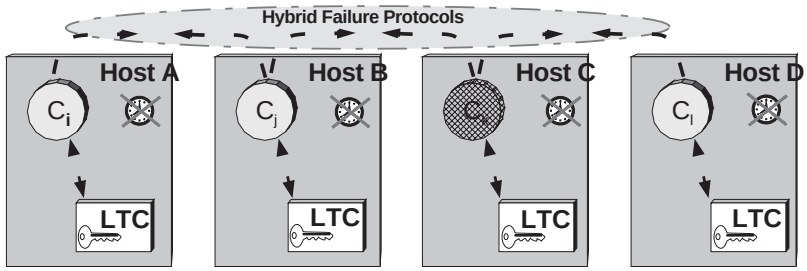

Figure 4. Fail-controlled with local trusted components

In the example, component C_k may be arbitrarily malicious, either because the component itself

or host C has been tampered with, or because an intruder in the communication system simulates that behavior. However, unlike the fail-uncontrolled strategy, the impact of this behavior on the other components (i.e., error propagation) may be limited, if the protocol makes components perform certain checks and validations with the LTC (for example, signature validation), which will prevent C_k from causing certain failures in the value domain (for example, forging). An additional proviso must be made: since the host environment is untrusted, IPC between a component and its LTC may be interfered with, though in a controlled way. For example, if host B is contaminated, component C_j may behave erroneously, but protocols can be designed in a way that prevents C_j from behaving in an arbitrary (e.g. Byzantine) way towards the other hosts.

This strategy is followed in the construction of the MAFTIA authorization service, described in Section 7. Components run distributed fault-tolerant authorization protocols based on capabilities that express the access control for objects. These protocols run among the authorization server replicas and the hosts running a MAFTIA application. Given the criticality of the authorization service, it is also worthwhile noting that the trust put on the Java Card LTC for this application is not absolute, in the sense that the higher-level protocols are ready to cope with the possibility of subversion of some Java Card modules and still ensure globally correct operation of the service. This is an excellent example of the innovative approach we take to trustworthy computing: components are trusted to the extent of their trustworthiness.

4.0.7 Fail-controlled with distributed trusted components

The "fail-controlled with distributed trusted components" strategy amplifies the scope of trustworthiness of the local component support, by making it distributed. As such, certain global actions can be trusted, despite a generally malicious communication environment. This strategy is implemented in MAFTIA through the TTCB (Trusted Timely Computing Base), which builds trust on global (distributed) time-related and security-related properties (such as global time, distributed durations, block agreement). One main impact of relying on the TTCB is that timed behavior can be supported globally in an intrusion-resilient way, as suggested by the warped clocks in Figure 5: the system is assumed to be *partially synchronous*, that is, anywhere in the interval ranging from time-free to fully synchronous, depending on the environment. This strategy assumes, as for the preceding strategies, that the hosts and communication environment are not trusted.

The distributed trusted component (DTC) is implemented by the local TTCBs interconnected by a control network. As with the "fail-controlled with local trusted components" strategy, in order for a protocol to be able to provide useful service, it has to cope with a hybrid of arbitrary and fail-silent behavior, depending on whether a component is interacting with the other components or with the TTCB. Consider the example of Figure 5, where again component C_k or host C may be arbitrarily malicious. Like the "fail-controlled with local trusted components" strategy, the impact of the faulty behavior of

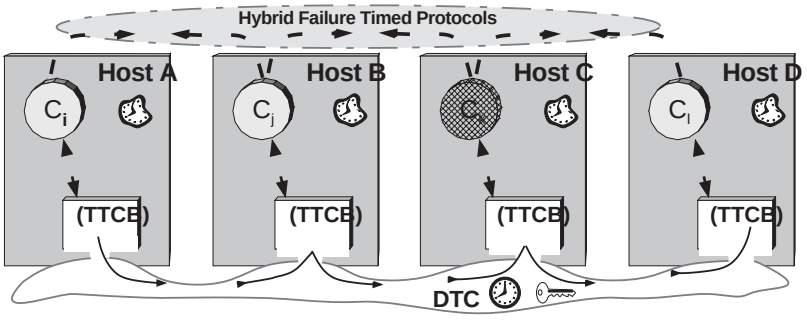

Figure 5. Fail-controlled with distributed trusted components

these components may be limited by enforcing certain validations with the local TTCB. However, the fact that the TTCBs are interconnected and can exchange information and perform agreement in a secure way — through the control channel — further limits the potential damage of malicious behavior: the DTC 'knows' directly what each of the components in different hosts 'say', unlike the solution with LTCs, where an LTC only 'knows' what a remote component 'says', through the local component. To achieve this, the TTCB allows the set-up of secure channels with any local component, and offers a low-level block consensus primitive. For example, components C_i through C_l could set up secure IPC with the TTCB, through which they would run such a consensus as part of the execution of some protocol.

The other relevant aspect of the TTCB strategy is time. The TTCB supports timed behavior in an intrusion-resilient way. As discussed in Section 3, timed systems are fragile in that timing assumptions can be manipulated by intruders. The TTCB supplies constructs that enable protocols to tolerate this class of intrusions. These are obviously related to the trusted time-related services briefly described earlier, namely absolute time, duration measurement and timing failure detection. As suggested in Figure 5, the TTCB DTC is a fully synchronous subsystem. It supplies its services to the payload system, which can have any degree of synchronism, as suggested by the warped clock. The TTCB does not make the payload system "more synchronous", but allows it to take advantage of its possible synchronism, in the presence of faults, both accidental and malicious. As such, the TTCB can assist an application running on the payload system to determine useful facts about time: for example, be sure it executed something on time; measure a duration; determine it was late doing something, etc. Then, the payload system, despite being imperfect (it suffers timing faults, some of which may result from attacks), can react (implement fault-tolerance mechanisms) based on reliable information about the presence or absence of errors (provided by the TTCB at its interface).

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

Several of the MAFTIA middleware protocols follow the "fail-controlled with TTCB" strategy. These protocols are group-oriented, deterministic, and can provide timeliness guarantees. The MAFTIA middleware provides different qualities of service in this timed profile by composing several micro-protocols on top of basic unreliable multicast. For example, this is the way in which reliable multicast and atomic multicast protocols described in Section 6 are achieved.

5 Byzantine Agreement: the arbitrary approach

As discussed before, an established way for enhancing the fault tolerance of a server is to distribute it among a set of servers and to use replication algorithms for masking faulty servers. Thus, no single server has to be trusted completely and the overall system derives its integrity from a majority of correct servers.

In this section, we describe a configuration of the MAFTIA architecture for distributing trusted services among a set of servers that guarantees liveness and safety of the services despite some servers being under control of an attacker or failing in arbitrary malicious ways. In this configuration, the system model does not include timing assumptions and is characterized by a static set of servers with point-to-point communication and by the use of modern cryptographic techniques. Trusted applications are implemented by deterministic state machines replicated on all servers and initialized to the same state. Client requests are delivered by an atomic broadcast protocol that imposes a total order on all requests and guarantees that the servers perform the same sequence of operations; such an atomic broadcast can be built from a randomized protocol to solve Byzantine agreement. We use efficient and provably secure agreement and broadcast protocols that have recently been developed.

In the first part of this section, we provide a detailed discussion of these assumptions, compare them to related efforts from the literature, and argue why we believe that these choices are adequate for trusted applications in an Internet environment. In the second part, a brief overview of the architecture and protocols is given. Our main tool is a protocol for atomic broadcast, which builds on reliable broadcast and multi-valued Byzantine agreement in an asynchronous network.

5.1 Model

In our model, the system consists of a static set of n servers, of which up to t may fail in completely arbitrary ways, and an unknown number of possibly faulty clients. All parties are linked by asynchronous point-to-point communication channels. Without loss of generality we assume that all faulty parties are controlled by a single adversary, who also controls the communication links and the internal clocks of all servers. The adversary is an arbitrary but computationally bounded algorithm. Faulty parties are called *corrupted*, the remaining ones are called *honest*. Furthermore, there is a trusted dealer that generates and distributes secret values to all servers once and for all, when the system is initialized. The system can process a practically unlimited number of requests afterwards.

This model falls under the impossibility result of Fischer, Lynch, and Paterson [23] of reaching consensus by deterministic protocols. Many developers of practical systems seem to have avoided this model in the past for that reason and have built systems that are weaker than consensus and Byzantine agreement. However, Byzantine agreement can be solved by randomization in an expected constant number of rounds only [13]. Although the first randomized agreement protocols were more of theoretical interest, some practical protocols have been developed recently. For example, the randomized agreement protocol of [10] is based on modern, efficient cryptographic techniques with provable security and withstands the maximal possible corruption.

In our system, we use Byzantine agreement as a primitive for implementing atomic broadcast, which in turn guarantees a total ordering of all delivered messages. Atomic broadcast is equivalent to Byzantine agreement in our model and thus considerably more expensive than reliable broadcast, which only provides agreement of the delivered messages, but no ordering (see Section 5.2).

Below we elaborate on the three key features of our model: cryptography, asynchronous communication, and a static server set.

Cryptography. Cryptographic techniques such as public-key encryption schemes and digital signatures are crucial already for many existing secure services. For distributing a service, we need distributed variants of them from *threshold cryptography*.

Threshold cryptographic schemes are non-trivial extensions of the classical concept of secret sharing in cryptography. Secret sharing allows a group of n parties to share a secret such that t or fewer of them have no information about it, but $t+1$ or more can uniquely reconstruct it. However, one cannot simply share the secret key of a cryptosystem and reconstruct it for decrypting a message because as soon as a single corrupted party knows the key, the cryptosystem becomes completely insecure and unusable.

A *threshold public-key cryptosystem* looks similar to an ordinary public-key cryptosystem with distributed decryption. There is a single public key for encryption, but each party holds a *key share* for decryption (all keys were generated by a trusted dealer). A party may process a decryption request for a particular ciphertext and output a decryption share together with a proof of its validity. Given a ciphertext resulting from encrypting some message and more than t valid decryption shares for that ciphertext, it is easy to recover the message. A threshold cryptosystem must be secure against adaptive chosen-ciphertext attacks [50], which means that the adversary cannot obtain any information from a ciphertext unless at least one honest server has generated a decryption share.

In a *threshold signature scheme*, each party holds a *share* of the secret signing key and may generate shares of signatures on individual messages upon request. The validity of a signature share can be verified for each party. From $t+1$ valid signature shares, one can generate a digital signature on the message that can later be verified using the single, publicly known signature verification key. In a secure threshold signature scheme, it must be infeasible for the adversary to produce $t+1$ valid signature shares that cannot be combined to a valid signature and to output a valid signature on a message for which *no*

honest party generated a signature share.

Another important cryptographic algorithm is *threshold coin-tossing scheme*, which provides a source of unpredictable random bits that can be queried only by a distributed protocol. It is the key to circumventing the FLP impossibility result [23] and used by the randomized Byzantine agreement protocol.

Threshold-cryptographic protocols have been used for secure service replication before, e.g., by Reiter and Birman [46]. However, a major complication for adopting threshold cryptography to our asynchronous distributed system is that many early protocols are not robust and that most protocols rely heavily on synchronous broadcast channels. Only very recently, non-interactive schemes have been developed that satisfy the appropriate notions of security, such as the threshold cryptosystem of Shoup and Gennaro [50] and the threshold signature scheme of Shoup [49]. Both have non-interactive variants that integrate well into our asynchronous model.

No Timing Assumptions. We do not make any timing assumptions and work in a completely asynchronous model. Asynchronous protocols are attractive because the alternative is to specify timeout values, which is very difficult when protecting against arbitrary failures that may be caused by a malicious attacker.

It is usually much easier for an intruder to block communication with a server than to subvert it. Prudent security engineering also gives the adversary full access to all specifications, including timeouts, and excludes only cryptographic keys from her view. Such an adversary may simply delay the communication with a server longer than the timeout and the server appears faulty to the others.

Time-based failure detectors [16] can easily be fooled into making an unlimited number of wrong failure suspicions about honest parties like this. The problem arises because one crucial assumption underlying the failure detector approach, namely that the communication system is stable for some longer periods when the failure detector is accurate, does not hold against a malicious adversary. A clever adversary may subvert a server and make it appear working properly until the moment at which it deviates from the protocol — but then it may be too late.

Of course, an asynchronous model cannot guarantee a bound on the overall response time of an application. But the asynchronous model can be seen as an elegant way to abstract from time-dependent peculiarities of an environment for proving an algorithm correct such that it satisfies liveness and safety under *all* timing conditions. By making no assumption about time at all, the coverage of the timing assumption appears much bigger, i.e., it has the potential to be justified in a wider range of real-world environments. For our applications, which focus on the security of trusted services, the resulting lack of timeliness seems tolerable.

Static Server Set. Distributing a trusted service among a static set of servers leverages the trust in the availability and integrity of each individual server to the whole system. In our model, this set remains fixed during the whole lifetime of the system, despite observable corruptions. The reason is that there

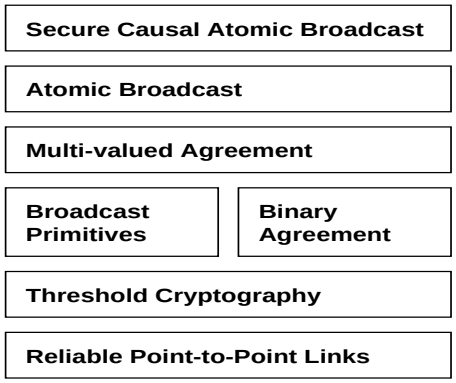

Figure 6. Asynchronous communications protocols for static groups.

are no protocols to replace corrupted servers in a secure distributed way, since all existing threshold-cryptographic protocols are based on fixed parameters (e.g., n and t) that must be known when the key shares are generated.

The alternative to a static server set is to remove apparently faulty servers from the system. This is the paradigm of view-based group communication systems in the crash-failure model [41]. They offer resilience against crash failures by eliminating non-responding servers from the current view and proceeding without them to the next view. Resurrected servers may join again in later views. But with the partial exception of Rampart [45], there is no group communication system that uses views and tolerates arbitrary failures (also Rampart cannot tolerate an attacker that has access to the failure detector).

The problem with Byzantine faults is that a corrupted server cannot be resurrected easily because the intruder may have seen all its cryptographic secrets. With the use of specialized "proactive" protocols [12], one could in principle achieve this by refreshing all key shares periodically. But such proactive cryptosystems for asynchronous networks have only recently been developed [8], after the formulation of this architecture, and further work would still be needed to build a fully asynchronous system with dynamic groups that tolerates Byzantine faults.

5.2 Secure Asynchronous Agreement and Broadcast Protocols

This section presents a short overview of the agreement and broadcast protocols used in this architecture configuration. Detailed descriptions can be found in related papers [10, 9, 11].

We need protocols for basic broadcasts (reliable and consistent broadcast), atomic broadcast, and secure causal atomic broadcast; they can be described and implemented in a modular way as follows, using multi-valued Byzantine agreement and randomized binary Byzantine agreement as primitives, as shown in Figure 6.

Byzantine agreement requires all parties to agree on a binary value that was proposed by an honest party. The protocol of Cachin et al. [10] follows the basic structure of Rabin's randomized protocol [42],

which is to check if the proposal value is unanimous and to adopt a random value, called a *common random coin*, if not. It terminates within an expected constant number of asynchronous rounds and uses a robust cryptographic threshold coin-tossing protocol, whose security is based on the so-called Diffie-Hellman problem. It requires a trusted dealer for setup, but can process an arbitrary number of independent agreements afterwards. Threshold signatures are further employed to decrease all messages to a constant size.

The other agreement primitive is *multi-valued Byzantine agreement*, which provides agreement on values from large domains. Multi-valued agreement requires a non-trivial extension of binary agreement. The difficulty in multi-valued Byzantine agreement is how to ensure the "validity" of the resulting value, which may come from a domain that has no a priori fixed size. Our approach to this is a new, "external" validity condition, using a global predicate with which every honest party can determine the validity of a proposed value. The protocol guarantees that the system may only decide for a value acceptable to honest parties. This rules out agreement protocols that decide on a value that no party proposed. Our implementation of multi-valued Byzantine agreement uses only a constant expected number of rounds; details can be found in [9].

A basic broadcast protocol in a distributed system with failures is *reliable broadcast*, which provides a way for a party to send a message to all other parties. Its specification requires that all honest parties deliver the same set of messages and that this set includes all messages broadcast by honest parties. However, it makes no assumptions if the sender of a message is corrupted and does not guarantee anything about the order in which messages are delivered. The reliable broadcast protocol of our architecture is an optimization of the elegant protocol by Bracha and Toueg [7]. We also use a variation of it, called *consistent broadcast*, which is advantageous in certain situations. It guarantees uniqueness of the delivered message, but relaxes the requirement that all honest parties actually deliver the message — a party may still learn about the existence of the message by other means and ask for it. Our implementation relies on non-interactive threshold signatures, which reduces the communication compared to previous implementations.

An *atomic broadcast* guarantees a total order on messages such that honest parties deliver all messages in the same order. Any implementation of atomic broadcast must implicitly reach agreement whether or not to deliver a message sent by a corrupted party and, intuitively, this is where Byzantine agreement is needed. The basic structure of our protocol follows the atomic broadcast protocol of Chandra and Toueg [16] for the crash-failure model: the parties proceed in global rounds and agree on a set of messages to deliver at the end of each round, using multi-valued agreement. For agreement every party proposes the messages that it wants to deliver in the current round. The agreement protocol decides on a list of messages and all messages in it are delivered according to a fixed order. The external validity condition ensures that all messages that are agreed-on list are appropriate for the current round. Details of this protocol are in [9].

A *secure causal atomic broadcast* is an atomic broadcast that also ensures a causal order among all

broadcast messages, as put forward by Reiter and Birman [46]. It can be implemented by combining an atomic broadcast protocol that tolerates a Byzantine adversary with a robust threshold cryptosystem. Encryption ensures that messages remain secret up to the moment at which they are guaranteed to be delivered. Thus, client requests to a trusted service using this broadcast remain confidential until they are scheduled and answered by the service. The threshold cryptosystem must be secure against adaptive chosen-ciphertext attacks to prevent the adversary from submitting any related message for delivery, which would violate causality in our context. Maintaining causality is crucial in the asynchronous environment for replicating services that involve confidential data. Our protocol for secure causal atomic broadcast follows the basic idea of Reiter and Birman's protocol. But because we use our atomic broadcast protocol and the non-interactive threshold cryptosystem of Shoup and Gennaro [50], we obtain the first provably secure implementation of secure causal atomic broadcast in an asynchronous network with Byzantine faults.

All our broadcast and agreement protocols work under the optimal assumption that $n > 3t$.

6 Reliable multicast: using trustworthy components

This section discusses MAFTIA architecture configurations following the strategy based on distributed trusted components (DTC). We solve a reliable multicast problem to exemplify the theory and the principles of construction of this kind of protocols.

The properties and correctness discussion of our protocols rely on the wormholes model, which postulates the existence of enhanced subsystems (wormholes) capable of providing a few simple privileged services to other components, with "good" properties otherwise not guaranteed by the "normal" weak environment in a distributed system [56]. For example, they can provide timely or secure functions in, respectively, asynchronous environments or systems with Byzantine failures. In MAFTIA the wormholes metaphor is materialized by the *Trusted Timely Computing Base (TTCB)* introduced in Section 3, a DTC providing a few timeliness- and security-related functions. Protocols built with a wormhole are run in a part of the system that might experience arbitrary delays or failures (asynchronous Byzantine environment). However, during their execution, they can call the wormhole services to perform correctly (small) crucial steps. In contrast to the rest of the system, the services only return trustworthy results. We say that such protocols are "wormhole-aware".

6.1 Model

Figure 7 shows a representation of the system in this configuration. Each node contains the typical software layers such as the operating system and runtime environments, and an extra component, the TTCB wormhole. The wormhole is distributed, with a local part in each node and a control channel. The local parts, or *local wormholes*, are computational components with activity, and the *control channel* is a private communication channel or network. The multicast protocol is executed by a group $P =$

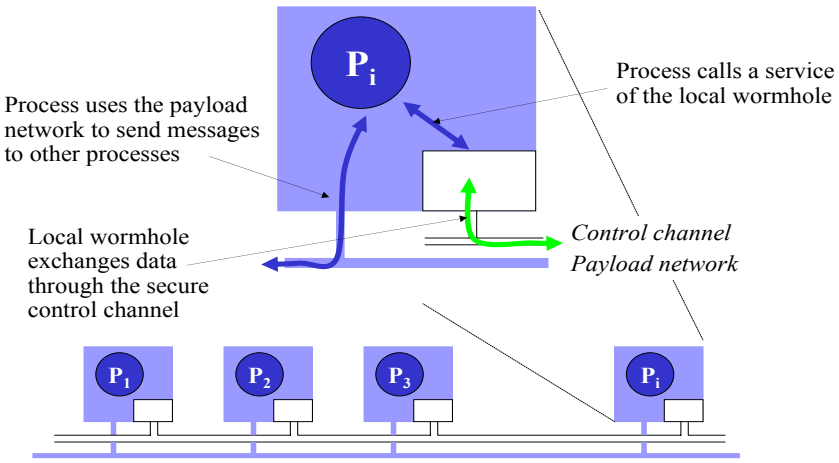

Figure 7. Architecture with a TTCB (payload subsystem is displayed in dark and the TTCB in white)

$\{p_1, ..., p_n\}$ of n processes. Processes run outside the wormhole and they communicate by sending messages through the payload network. At certain points of their execution, they can, however, request some services of the wormhole by calling its interface.

With the exception of the wormhole, the system is assumed to be asynchronous. Consequently, there can be no assumptions about the relative speed of processes, no bounds on message delivery delays (on the payload network), and no bounds on the invocations of wormhole services (since they are initiated and terminated in the asynchronous part of the system). The wormhole is assumed to be synchronous and capable of timely behavior. This means that once a request arrives at the wormhole interface, it will take a well defined interval until that answer is available at the same interface. In practical implementations these synchrony guarantees can be ensured because the wormhole has complete control over all resources in the node that are needed to perform its tasks, including the control channel (for details see [52, 14, 19]).

The payload part of the system, which includes the processes, can suffer from Byzantine faults. For instance, processes can stop working, skip some steps in the protocol, give invalid information to the wormhole or other processes, or collude with other malicious processes in an attempt to break the protocol. Byzantine faults can also affect the communication through the payload network and the service calls to the local wormhole. We assume that the payload network has an associated *omission degree* (Od), which implies that no more than Od messages are corrupted/lost in a reference interval of time. By making this assumption, a message only needs to be retransmitted $Od + 1$ times to ensure its reception in absence of attacks. In a Byzantine setting, however, for sufficiently strong adversaries one can envision attacks that will corrupt more than $Od + 1$ successive retransmissions. Whenever this happens, we take the approach of considering the receiver process as faulty (which is indeed the end result, since that process is unable to communicate). Nevertheless, the reader should notice that Od is just a parameter of the protocol: conservatively large values of Od basically simulate a reliable channel. Incidentally, if a process is systematically prevented from calling the local wormhole, then it will also be considered

as faulty.

The wormhole subsystem, which includes the control network, is assumed to fail only by crashing. Therefore, a local wormhole either provides its services as expected or it simply stops running. This assumption should hold even if malicious adversaries manage to attack and compromise a node with a local wormhole (for implementation details see [19]).

6.2 Wormhole services and interface

A wormhole provides a small number of services that can be accessed through calls to its local interface. Distinct protocols can utilize different services in different ways. However, in all cases, protocols are designed to run on the payload subsystem and use the wormhole infrequently, imposing a comparatively small load on it. The TTCB (Trusted Timely Computing Base), is just one example of wormhole, the one implemented in MAFTIA, whose most important services are briefly described ahead (a detailed description can be found in [35]).

The *Local Authentication service* (this is a component-oriented, rather than a high-level authentication service) makes the necessary initializations and authenticates the local wormhole component before the process. A timestamp with the current global time is returned by the *Timestamping service*. The *Trusted Block Agreement service* applies an agreement function to a set of values proposed by the processes and returns a value. By using different functions, this service can be configured to deliver results with diverse characteristics. Perhaps a good way to understand the service is through the description of its interface parameters:

tag,error ←TTCB_propose(eid, elist, tstart, decision, value)

value,prop-ok,prop-any,error ←TTCB_decide(eid, tag)

A process calls $TTCB\_propose$ to propose a value. The parameters have the meaning: eid is the unique identification of a process before the wormhole, obtained using the Local Authentication Service. $elist$ is a list with the $eids$ of the processes involved in the agreement. $tstart$ is a timestamp and corresponds to the latest instant when the agreement will start (it might be initiated earlier if all proposals arrive before $tstart$). A proposal made after $tstart$ is rejected and an error is returned. *decision* defines the agreement function (the TTCB offers a limited set). $value$ is the proposed value, and it can only have a small number of bytes (20 in the current implementation). The function returns an $error$ code and a unique identifier of this agreement, called the tag. Processes call *TTCB_decide* to get the result of the agreement. The result is a record with four fields: the decided *value*, a mask *prop-ok* with one bit set per each process that proposed the decided value, a mask *prop-any* with one bit set per process that proposed any value, and an *error* code.

6.3 Designing wormhole-aware protocols

We use a reliable multicast protocol as an example to illustrate the principles of building wormhole-aware protocols (more details can be found in [18]). A reliable multicast protocol guarantees that all correct processes deliver the same messages, and if a correct sender transmits a message then all correct processes deliver this message. No assurances, however, are provided about the order of message delivery. Formally, a reliable multicast protocol has the following properties [26] [4].

- *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 $sender(M)$.

One achievement related to our model is that the protocol requires that, out of a total of n processes, no more than $f = n - 2$ processes are allowed to fail (the problem is of obviously little interest with less than two correct processes). The asynchronous Byzantine system model augmented with the TTCB wormhole allows us to lower the known limit of $f = (n - 1)/3$ processes [18].

Basic principles. Wormhole-aware protocols can be depicted as running on a dual space-time diagram (see Figure 8): the payload subsystem's, seen in evidence in the figure; and the TTCB subsystem's, whose timelines are collapsed in the thick gray bar for simplicity. A correct use of the wormholes principle mandates that most of the protocol execution takes place in the payload subsystem. The wormhole services are only invoked when there is an obvious trade-off between what is obtained from the wormhole service and the complexity/cost of implementing it in the payload subsystem (e.g., related with hard reliability, synchrony, or security requirements). This will become evident from the examples to come.

As usual, the protocols starts with the sender multicasting a message through the payload channel. Since the sender might be malicious or the network might be attacked or have omission failures, the protocol needs to ensure the reception of the message by all correct processes and the integrity of the message contents.

In asynchronous Byzantine environments this may entail some complexity and/or delay [44, 28]. The wormhole has thus an opportunity to come into play: the sender and all recipients send a hash of the message just sent/received, to the wormhole, which runs an agreement on the hashes in its protected environment, returning to all the sender's hash as result. If all goes well, processes see that: all proposed

[4] 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).

the same hash; and that the agreed hash corresponds to the message received. Thus, they terminate at the end of this phase, in an extremely fast manner.

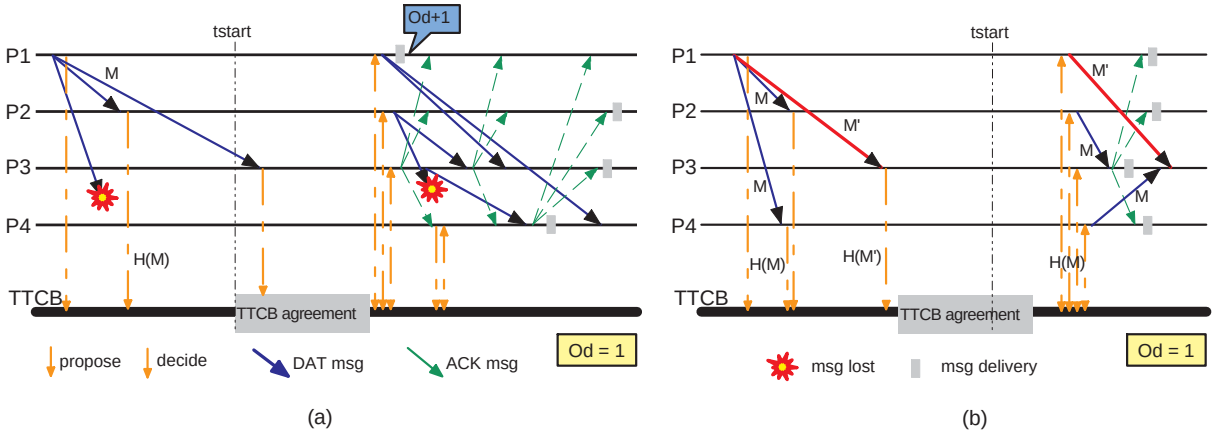

Figure 8. Reliable Multicast with a Wormhole: (a) omissions and delays; (b) malicious sender.

Detecting and recovering from errors. If this condition is not verified, then either one or more processes did not get the correct message or some of them were late when proposing to the wormhole (after $tstart$). Such a situation is depicted in Figure 8a, and it requires a second phase: P_3 is going to propose late and thus be rejected; P_4 suffers an omission. Detecting these errors is another difficult task in asynchronous Byzantine environments, and here we see another contribution of the wormhole: reliable error detection (performed concurrently with the agreement on the hash). Processes detect the errors immediately, because they spot "holes" in the *prop-ok* and *prop-any* masks (see Section 6.2), respectively for processes that proposed wrongly or did not propose at all.

In the second phase, processes try to remedy for the above-mentioned "holes", using classical retransmission and forwarding techniques. As shown in Figure 8a, processes retransmit the message until either all acknowledge the arrival of the message or the $Od + 1$ threshold is reached. The 'bounded omission degree' technique for synchronous environments [54] can be used here for termination in an asynchronous environment, due to the synchronous distributed error detection channel provided by the TTCB wormhole.

Each multicast is retransmitted at most $Od + 1$ times in order to resist a number of network failures ($Od = 1$ in the setting of the example). After a few forwardings and acknowledgements, we see that all processes end-up converging and deliver the message. In terms of protocol principle, the reader should see these operations as recovery measures that end-up asserting the missing bits in the masks. Figure 8b depicts the more serious situation where the sender is Byzantine: it sends a different message M' to P_3, who obviously proposes hash $H(M')$ and is deemed wrong. The remedy is used again: correct recipients forward M to P_3, who asserts its bit in its own *prop-ok* mask and acknowledges. The other recipients do the same in reaction to the acknowledgement, and all deliver.

Payload-wormhole synchronization. The role of $tstart$ deserves mention, since it can be generalized to other protocols as well. $tstart$ reveals an interesting way of performing synchronization between the asynchronous payload and the synchronous TTCB wormhole protocols.

To the latter, such a quantity is a real time instant or interval in the global timeline maintained by all wormhole modules. It allows wormhole protocols to perform essentially all timed operations. To the former, the time-free payload protocols, it is merely an integer quantity. These 'synchronizers' can be used as constants or variables, and establish a sound basis for decoupling the logic and timing aspects of the design of protocols [51]. Time-free modules can safely synchronize with each other by exchanging and/or agreeing on these values, adding quantities to them, and so forth, since all timing aspects (including timing failure, a.k.a timeout, detection) are dealt with in the wormhole. These values can be bound to real time when desired, by using the Timestamping service [53].

In this particular case, $tstart$ is set to the current time plus a delay. To prevent the service execution from being maliciously postponed, the Trusted Block Agreement service rejects proposals that arrive after $tstart$. Therefore, the selection of the $tstart$ value presents a tradeoff: if it is too large, the first phase may take longer than what is required; if the value is too small, a correct recipient may not receive the message before $tstart$ and the second phase will have to be executed unnecessarily (i.e., the opportunity to terminate the protocol early is lost). Note however that the value of $tstart$ only affects performance, and not safety or liveness. A good heuristics for selecting the delay is to make it proportional to an estimate of the message transmission time. Incidentally, in another work we have studied the dependable dynamic adaptation of timing parameters such as message transmission times, in systems using wormholes [15].

Acknowledgement protection. The acknowledgements, as the data messages, have to be protected from forgeries. To accomplish this task, we use a vector of Message Authentication Codes (MACs) through the payload, instead of relying on the services of the wormhole. This is a good example of a situation where the trade-off for using the wormhole does not occur: what is going to be used potentially massively (acknowledgements), and could be easily and efficiently achieved through the payload (MACs), should not go through the wormhole.

A MAC is a cryptographic checksum obtained with a hash function and a symmetric key [32]. Although MACs are not as powerful as signatures based on public-key cryptography, they are sufficient for our needs, and more importantly, they are several orders of magnitude faster to calculate. Since acknowledgements are multicasted to all processes, they do not take a single MAC but a vector of MACs with an entry per process.

7 Authorization

Authorization aims to ensure that only legitimate actions are carried out in the system, or, equivalently, to prevent illegitimate actions from being carried out. As such, authorization, and its implementation by access control mechanisms, participates in error detection (by detecting attempts to run illegitimate actions) and in error confinement (by preventing illegitimate actions), whether these errors are due to accidental faults or attacks. Authorization is thus of tremendous importance for Internet applications.

Currently, the most common authorization scheme used on the Internet is based on the client-server paradigm: a server satisfies or rejects client requests at its discretion, according to what it knows about the client (e.g., the identity claimed by the client, history of previous transactions, etc.). Unfortunately, the client-server model is not rich enough to cope with complex transactions involving more than two participants. For example, an electronic commerce transaction requires usually the cooperation of a customer, a merchant, a credit card company, a bank, a delivery company, etc. Each of these participants has different interests, and thus distrusts the other participants. Moreover, such a model is necessarily privacy intrusive, since it enables the server to record a lot of personal information about clients: identity, usual IP address, postal address, credit card number, purchase habits, etc.

MAFTIA proposes a new authorization scheme that can grant fair rights to each participant, while distributing to each one only the information strictly needed to execute its own task, i.e., a proof that the task has to be executed and the parameters needed for this execution, without unnecessary information such as participant identities. This scheme is based on two levels of access control:

An *authorization server* is in charge of granting or denying rights for transactions involving several participants; if a transaction is authorized, the authorization server distributes authorization proofs (i.e., capabilities) for all the elementary operations that are needed to carry it out.

On each participating host, a *reference monitor* is responsible for fine-grain authorization, i.e., for controlling the access to all local resources and objects according to the capabilities that accompany each request. To enforce hack-proofing of such reference monitors on off-the-shelf computers connected to the Internet, critical parts of the reference monitor are based on a MAFTIA trusted component, the Java Card Module, as described in Section 3.

7.1 Authorization Server

In [37], a generic authorization scheme had been proposed for distributed object systems. In this scheme, an application can be viewed at two levels of abstraction: composite operations and method executions. A composite operation corresponds to the coordinated execution of several object methods towards a common goal. For instance, printing file F3 on printer P4 is a composite operation involving the execution of a *printfile* method of the spooler object attached to P4, which itself has to request the execution of the *readfile* method of the file server object managing F3, etc. In the MAFTIA context, composite operations can be assimilated to transactions.

A request to run a transaction is authorized or denied by an authorization server, according to *symbolic rights* stored in an access control matrix managed by the authorization server. More details on how the authorization server checks if a transaction is to be granted or denied are given in [36] and [2]. If the request is authorized, capabilities are created by the authorization server for all the method executions needed to perform the transaction. These capabilities are simple method capabilities if they are used directly by the object requesting the execution of the transaction, i.e., used by this object to directly call another object's methods. Alternatively, the capabilities may be indirect capabilities or *vouchers*, if they cannot be used by the calling object but must be delegated to another object that will invoke other object methods to participate in the transaction. In fact, the notion of transaction is recursive, and a voucher can contain either a method capability or the right to execute a nested transaction.

This delegation scheme is more flexible than the usual "*proxy*" scheme by which an object transmits to another object some of its access rights for this delegated object to execute operations on behalf of the delegating object. Our scheme is also closer to the "*least privilege principle*", since it helps to reduce the privilege needed to perform delegated operations. For instance, if an object O is authorized to print a file, it has to delegate a *read-right* to the spooler object, for the latter to be able to read the file to be printed. To delegate this read-right with the proxy scheme, O must possess this read-right and could thus abuse this right by making copies of the file and distributing them. In this case, the read-right is a privilege much higher than a simple print-right. In our scheme, if O is authorized to print a file, O will receive a *voucher* for the spooler to read the file, and a capability to call the spooler. The voucher, by itself, cannot be used by O. With the capability, O can invoke the spooler and transmit the voucher to the spooler. The spooler can then use the voucher as a capability to read the file.

Since only transactions are managed by the authorization server, system security is relatively easy to manage: the users and the security administrators have just to assign the rights to execute predefined transactions, they do not have to consider all the elementary rights to invoke object methods. Moreover, since only one request has to be checked for each transaction, the communication overhead can be reduced.

The authorization server is a trusted third party (TTP), which could be a single point of failure, in case of both accidental failure or successful intrusion (including by a malicious administrator). To prevent this, with the MAFTIA authorization architecture, the authorization server is made fault- and intrusion-tolerant [2]: an authorization server is made of diversely-designed and implemented security sites, operated by independent persons. Faults and intrusions affecting security sites can be tolerated without degrading the service, as long as only a few security sites are affected.

In order to tolerate the failure of one or a small number of the sites composing the authorization server, two main protocols are used:

Mutual agreement: all non-faulty sites agree on the decision to grant or deny the authorization corresponding to a given request. This guarantees a correct decision as long as there is only a minority of faulty sites. In practice, the number f of faulty sites may have to be much less than half the total number

n of sites, depending on the fault assumptions. For instance, (n ¿ $3f$) must be guaranteed if Byzantine faults are to be taken into account.

Threshold signature: the capabilities and vouchers are globally signed by the authorization server, using a threshold signature scheme. Each of the sites composing the authorization server generates a signature share (depending on its own private key share) so that if at least t valid signature shares are available (t being the threshold), it is possible to combine these shares to generate a unique signature that can be verified with a global public key. This guarantees that if a capability (or a voucher) has a correct signature, the corresponding operation is indeed authorized (the signed capability cannot be forged, even by a cooperation of f faulty sites, as long as f is strictly less than the threshold t).

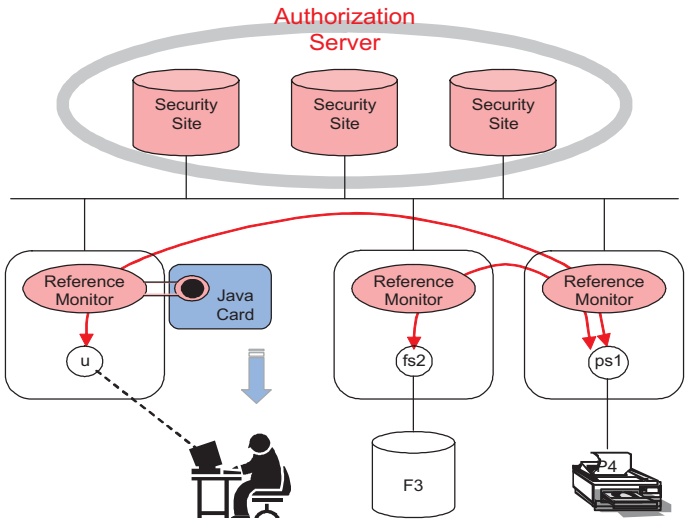

Figure 9. Authorization architecture.

These protocols are presented with more detail in [3]. The global architecture is given by Figure 9. The dialogue between a MAFTIA object and the authorization server is typically as follows (*see* Figure 10).

Object O asks the authorization server for the authorization to carry out an operation in the system. This operation may be the simple invocation of a particular method of a particular object O' or may be a transaction that requires the collaboration between several objects in the system.

In the first case, if object O is authorized to carry out the operation, it receives a capability, encrypted by the public key of the host where O' is located, and then signed using the threshold scheme described above. This capability will be presented and checked by the reference monitor located on the site of the invoked object O'.

In the second case, the user may receive several capabilities and vouchers. Capabilities are directly used by object O to invoke particular methods of particular objects, and are encrypted and signed as in the first case. Vouchers are not used by object O but are forwarded by object O to other objects that are involved in the execution of the transaction (e.g., a capability for O' to invoke a method m of an

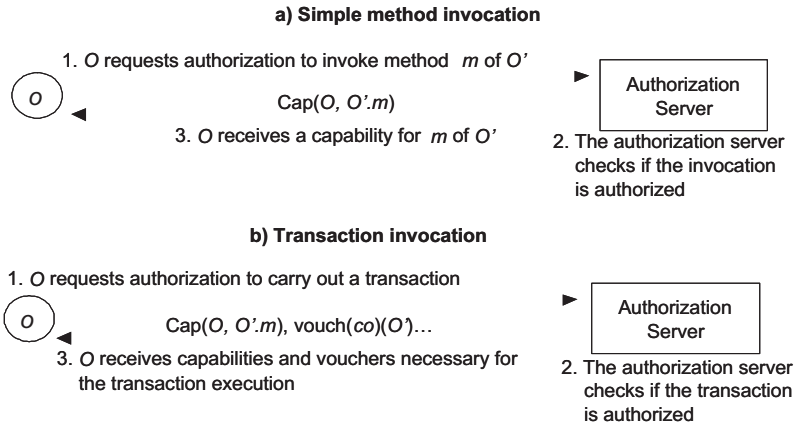

Figure 10. Protocol between a MAFTIA object and the authorization server.

object O'', as a part of the transaction). These vouchers will thus be transferred by object O to other objects, which will then execute their part of the transaction thanks to these vouchers. A voucher may be a capability (in which case they are encrypted and signed as above), or the right to execute a nested transaction (in which case the voucher is just signed). In the latter case, this voucher is a *token* that has to be presented to the authorization server to obtain directly all the authorization proofs needed to execute the nested transaction, without consultation of the access control matrix.

7.2 Reference Monitor

There is a reference monitor on each host participating in a MAFTIA-compliant application. The reference monitor is responsible for granting or denying local object method invocations, according to capabilities and vouchers distributed by the authorization server. In the context of wide-area networks (such as the Internet), the implementation of such a reference monitor is complicated since, due to the heterogeneity of connected hosts, it would be necessary to develop one version of the reference monitor for each kind of host. Moreover, since the hosts are not under the control of a global authority, there is no way to ensure that each host is running a genuine reference monitor, or the same version thereof. This is one reason we have chosen to implement them by using Java Cards.

We assume that any software, even that within an operating system or a JVM, can be copied and modified by a malicious user who possesses all privileges on a local host. In particular, on the Internet, any hacker can easily have these privileges on his own computer! The capability checks only provide assurance to *non-faulty* hosts, who can be sure that any remote request to execute a MAFTIA-application is genuine (if the capability is correct), and that a genuine MAFTIA request can only be executed on a host for which the capability is valid.

However, if a host is faulty or has been corrupted by a hacker, there is no assurance at all about the operations it executes locally. Nevertheless, we do trust the local Java Card Module to protect the

integrity of a corrupt host's private key. This means in particular that we exclude the possibility of a corrupt host being able to impersonate a non-faulty host since it is unable to sign messages correctly. Thus, the privileges gained by a hacker on a corrupt host give him no privilege outside that host unless the hacker is able to tamper with the local Java Card Module.

We consider the Java Card Module to be *sufficiently* tamperproof, as discussed in Section 3, in order to sufficiently delay the attacker's progress in corrupting further hosts. Global properties can thus be maintained by fault-tolerance mechanisms unless, say, more than f hosts are compromised in a given "window of vulnerability" [59]. The fact that the authorization scheme prevents a hacker from gaining privileges outside a corrupted host unless he successfully tampers with the Java Card Module means that the difficulty of violating such a global property is linear in f.

The capability checks carried out by the Java Card Module are based on strong cryptographic functions. Several cryptographic keys must be included in the Java Card:

PK_m, the MAFTIA public key. This key is associated to the MAFTIA private key SK_m, which is not stored in the Java Card. The Java object classes are signed off-line by this key SK_m, and this signature is checked at load time by the local JVM of the host[5], using PK_m stored in the Java Card.

SK_j, PK_j, a private/public key pair specific to the Java Card, thus specific to the host.

PK_{as}, the authorization server public key. This key is associated to all the private key shares of all the sites composing the authorization server.

Each capability is encrypted by the authorization server, using the public key PK_j of the site where the invoked object is located. Then the capability is signed by the authorization server with a threshold signature protocol. Consequently, the capability signature must first be verified using the authorization server's public key PK_{as}, and then decrypted (by the cryptographic functions of the Java Card) using the private key SK_j, which is stored only in the Java Card. Each access to a method of an object on a MAFTIA host is first intercepted by a *Dispatcher*, which is a Java object, located in the local JVM interfacing the Java Card. For each access to a local object method, the dispatcher checks if the invocation is carrying a capability, then sends this capability to the Java Card for verification. This verification corresponds to step 2 of Figure 11.

Other information can be stored in the Java Card, for instance for the authentication of the user owning the Java Card if the host is a personal workstation, or for the authentication of the administrator who has been assigned this Java Card if the host is a server. In the latter case, it may be possible to have several administrators for the same server, each administrator having his personal Java Card for this server, and all server administrator Java Cards sharing the same pair SK_j, PK_j. More details on the Java Card implementation of the reference monitor can be found in [21].

[5] Since version 1.2, the Java Development Kit includes software that allows classes to be signed and the signatures to be checked at load time.

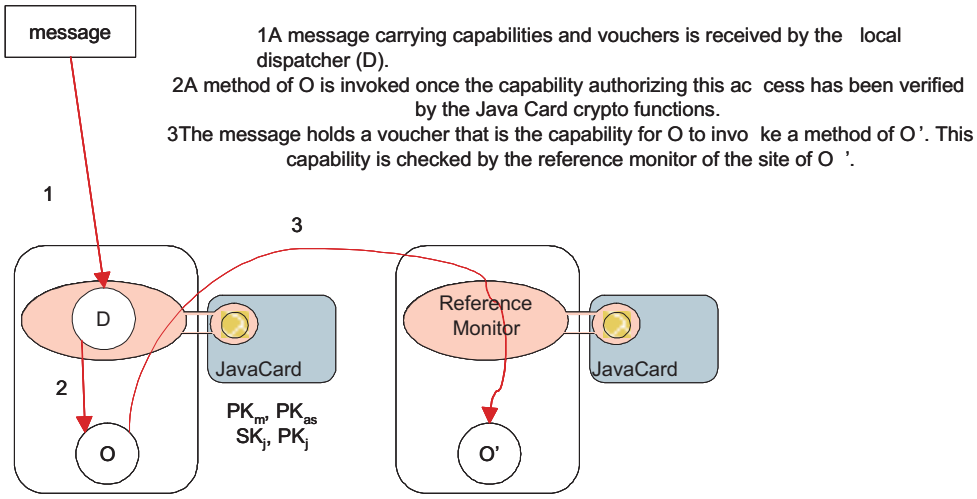

Figure 11. Example of a voucher corresponding to a capability.

8 Transactions: providing IT to applications

This section describes the MAFTIA transactional support service. The transactional support service is intended to support both applications built using the MAFTIA middleware and other activity support services, for example it can be used to guarantee the atomicity of updates to a replicated authorization server. The transaction support service appears to the user as a CORBA-style transaction service. Its intrusion tolerance is a non-functional property of the service implementation, transparent to the interface.

8.1 Overview

The MAFTIA transaction service provides a mechanism for implementing application-level intrusion tolerance and is itself intrusion-tolerant. To do this we apply the general MAFTIA architectural principle of distributing trust by replicating the servers implementing the transaction service and optionally the resource managers.

Our approach is to make use of standard group communication primitives, allow for heterogeneous resources, apply error compensation techniques to improve intrusion tolerance, to allow for multi-party transactions consider failure atomicity and allow recovery without reliance upon durable storage. This differentiates our work from approaches that make use of new or modified group communication primitives (for example, optimistic broadcast) [25, 47, 57]. Also, unlike other approaches, our focus is not on availability but on intrusion tolerance. This has resulted in us not being able to use techniques such as passive replication that are widely used by the database community. Passive replication is more efficient than active replication, and does not require deterministic replicas. However, the problem with adopting passive replication is its reliance on a leader-follower model. The updates occur at the leader and the followers are informed of the results. Whereas this is adequate in a crash-failure fault model, it is in-

appropriate for malicious faults, because the leader becomes a single point of failure: the leader can be corrupted and start sending corrupted updates to the leaders. By adopting an active replication approach we avoid this problem as there is no single point of failure and more that f members of the group must be corrupted before the group as a whole is compromised.

The approach that is closest to our is that of *GroupTransactions* [39] although our model of multi-party clients is different: our multiple parties are pre-existing and are not created within the context of a group transaction. Our model is also more general, since their system model assumes a LAN and does not explicitly consider malicious faults. However, they do address nested transactions whereas we only implement a flat transaction model.

8.2 Architecture

An overview of the transaction service architecture is shown in Figure 12. We have implemented the transaction service using the Appia framework [34]. Each major component and constituent Appia protocol layers are shown. As in a traditional distributed transaction service, the architecture is made up of clients, resource managers and transaction managers. Multiple clients interact with replicated transaction and resource managers. Our architecture differs from a traditional architecture in several ways: the transaction manager and resource manager components are replicated; each component uses intrusion-tolerant group communication protocols to provide intrusion tolerance; managers avoid the need for durable storage for recovery; and our service supports multi-party transactions.

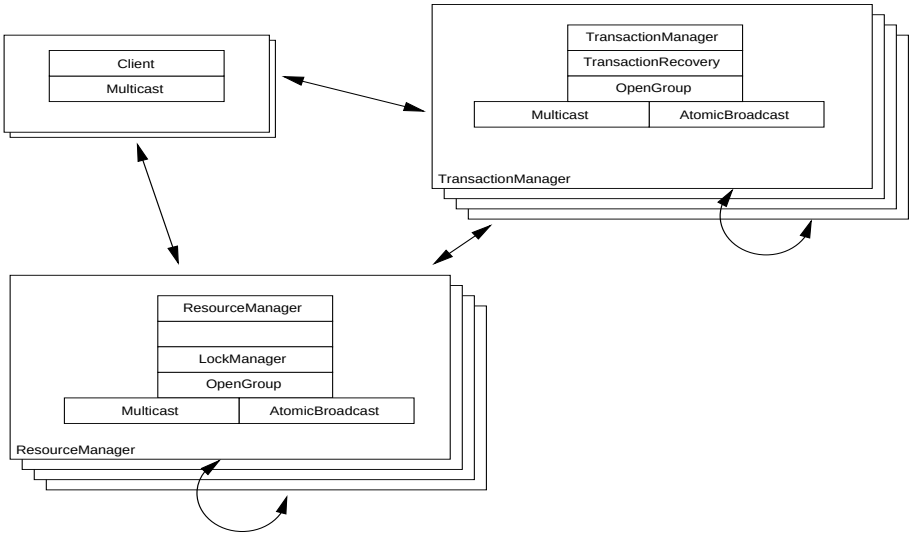

Figure 12. Overview of MAFTIA Transaction Service

Clients use the transaction manager to begin and end transactions and within the scope of a transaction the clients operate on resources via resource managers. There may be multiple clients participating in the same transaction or multiple clients participating in different transactions. The transaction manager

is primarily a protocol engine. It implements the two-phase commit protocol and recovery protocol. It also allows the creation of new transactions and the marking of transaction boundaries. In order to participate in transactions, resource managers are required to register themselves with the transaction manager. A resource manager is a wrapper for resources that allows the resource to participate in two-phase commit and recovery protocols coordinated by a transaction manager. The resource may or may not be persistent. In our implementation concurrency control is pessimistic but an optimistic scheme could also be implemented with minimal change to the interfaces of the resource manager.

Although appearing to clients as standard transaction and resource managers, the components of the MAFTIA transaction service are replicated. When a client makes a request it is processed by all the replicas and the client is delivered the result returned by the majority of replicas. To prevent an adversary manipulating the order of delivery of requests and therefore the outcome, we rely upon the MAFTIA intrusion-tolerant group communication service.

Assuming that failure can be reliably detected then a recovery mechanism is required to allow the resumption of transaction processing. Unlike in a traditional distributed transaction service, the local durable log cannot be used to recover because it may have been compromised. So, in the MAFTIA transaction service the recovery for transaction managers and resource managers relies upon a recovering replica querying the group to determine the state of the log.

We support multi-party transactions with a clear semantics on how clients share the decision for aborting or committing a transaction. In our model a single client begins a transaction, and passes the transaction identifier to other clients so that they can cooperate within the transaction scope too. Individual clients can unilaterally force a transaction abort but all clients must unanimously agree to attempt a transaction commit. These semantics are based upon those of Coordinated Atomic Actions (CAA) [58, 43] where participants must either agree on a normal or exceptional outcome, or abort the entire action. The general notion is to provide clear semantics for the termination of join action but to allow parties to share resources freely within the context of a transaction. Note that unlike CAA our model of multi-party transactions does not currently consider agreement upon exceptions.

8.3 Reliance on MAFTIA middleware

The MAFTIA transaction service both supports application-level intrusion tolerance by providing error confinement for multi-party interactions and is itself intrusion-tolerant because it provides correct service in the presence of compromised transaction and resource managers. Correct service is achieved through the application of the principle of error compensation that is implemented using active or “state machine” replication [40]. The transaction service is composed of replicated and diverse resource manager and transaction manager servers. We rely upon the MAFTIA middleware’s communication services to implement the replication. Therefore, in order for the transaction service to tolerate intrusions, we need the communication services to be intrusion tolerant.

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

As discussed above, we replicate transaction managers and resource managers. These form server groups that are distributed across sites. Server groups are a set of n servers, of which up to f may fail in completely arbitrary ways. All members of the service group handle requests and the majority result is returned to the user of the service. This means that as long as no more than f servers fail, the overall service remains trustworthy. To allow voting on results the servers are assumed to be deterministic. In the arbitrary model, $3f + 1$ servers are sufficient to provide correct service in the face of f expected failures. In the TTCB model, depending upon the implementation of atomic broadcast that is used then as few as $2f+1$ servers may be sufficient. Note that it is assumed that the groups have static membership and sufficient diversity of implementation, platform etc. to give assurance that the servers do not share a common failure mode.

Recovery in our implementation does not depend upon local durable storage as an attacker may compromise this. Instead, a recovering group member will contact other group members to determine what its state should be. Such an approach assumes that there is some mechanism that ensures that a recovering member can be successfully reinitialized without compromising the security of the group. For example, when using the asynchronous timing assumptions then recovery may require the trusted dealer to redistribute keys so that the security of the group is maintained.

Interacting with the transaction managers and resource managers are an unknown number of possibly faulty clients. Clients are outside our control and can be implemented in any way. Therefore they can fail in arbitrary ways. Currently we do not make clients intrusion-tolerant or the transaction service tolerant of misbehaving clients. For example, clients may block the progress of transactions or access to resources managed by resource managers. We have avoided using timeouts to resolve these problems as they introduce a vulnerability that could be exploited by an attacker.

8.4 Implementation

As the Appia framework was the standard implementation framework for the MAFTIA middleware then this simplified integration with the MAFTIA communication services. Because both the arbitrary (*cf.* Section 5) and TTCB-based (*cf.* Section 6) communication services were implemented using the same framework, the transaction service can run under either configuration. Integration with other frameworks would simply require the substitution of the AtomicBroadcast layer.

As future work, the existing transaction service could be integrated with the authorization service. This would enhance intrusion-tolerance, for example, for example participation in transactions could be restricted to trusted clients. Enhancing our current use of a transaction identifier by treating it as a capability could do this. In this model, capabilities for participating in transactions and accessing resources would be issued by the distributed authorization server, and checked by the transaction and resource managers.

On another tone, the transaction service could use the TTCB functionality directly, as now done by the communication protocols. For example, the TTCB provides a basic service that allows for consensus on a limited amount of state. Invoking this service directly instead of via the Atomic Broadcast protocols could provide considerably better performance, at the cost of tying the transaction service implementation to a particular intrusion tolerance strategy.

9 Conclusion

We presented the approach taken in MAFTIA to architect and build intrusion-tolerant systems, i.e., systems that are assumed to remain to some extent faulty and/or vulnerable and subject to attacks that can be successful, the idea being to ensure that the overall system nevertheless remains secure and operational, using notions pertaining to the generic 'tolerance' paradigm.

We started by revising the basic dependability concepts under a security-related perspective, incorporating specific security properties, fault classifications, and security methods. Under the light of these revised concepts, the term *trustworthiness*, often preferred when the focus is on malicious activity, is essentially synonymous to dependability, and has a powerful and precise meaning: it points to generic properties and not just security; it has a well-defined relationship with the notion of *trust*. This relation supports an important design principle in MAFTIA: a *trusted component* has a set of properties that are trusted *to the extent of* the component's trustworthiness.

In the course of developing the MAFTIA architecture and intrusion-tolerant middleware, we were faced with a multitude of challenges that we shared with the reader, since they are common to any endeavor in distributed, malicious-fault tolerant architectures. As a result, we devised new architectural constructs and algorithmic strategies. We introduced architectural hybridization as a means to substantiate the notion of 'trustworthy trusted component' in a malicious-fault environment. We devised programming models based on the modular use of trusted components, taking advantage of their fault prevention potential to recursively assist and augment the power of fault-tolerance mechanisms. We developed protocols that reason in terms of the availability of such trusted components, to achieve efficient operation whilst preserving resilience. On the alternative track of fail-arbitrary asynchronous environments, satisfying safety under any conditions for highly critical security uses, we devised provably secure protocols employing efficient cryptographic techniques with randomization. We proposed a new authorization scheme that overcomes privilege amplification or privacy violation problems in multi-

participant transactions, based on innovative access control protocols. Finally, we introduced replication and transaction control mechanisms built on top of the mentioned protocols, in an illustration of the recursive, component-based overall strategy for intrusion tolerance in MAFTIA explained in the paper. The rationale behind these protocols, whose algorithmics is detailed in other publications, was presented as a proof of concept of the several strategies to achieve intrusion tolerance in and with the MAFTIA architecture and middleware.

Finally, the notion of handling a wide set of faults encompassing intentional and malicious faults in order to preserve system properties (security or other), if successfully achieved, as we hope to have demonstrated, has two striking effects: (a) it leads us to *x-tolerant system frameworks*, common system design principles where "any fault set" can be handled, instead of (as presently) changing framework depending on the fault model and application; (b) it presents a great advance on the ability to design accidental fault-tolerant systems in complex and unpredictable settings (presently a research subject).

Acknowledgements

MAFTIA was a project of the IST programme of the European Commission, whose participants were: FCUL University of Lisboa, IBM Zurich Research Labs, LAAS-CNRS, Qinetiq, University of Newcastle upon Tyne, Universitt des Saarlandes. We wish to warmly thank all the members of the project teams, whose contributions were invaluable for the collective success of the project. The achievements of MAFTIA are not limited to architectural work. The MAFTIA portal[6], where the reader can find all publications of the project, details other very successful work strands, not only the formal verification of some of the protocols discussed here, but also, for example, application work on intrusion detection.

References

[1] J.-C. Laprie A. Avizienis and B. Randell. Fundamental concepts of dependability. In *Proceedings of the IEEE Third Information Survivability Workshop*, Boston, MA, USA, October 2000. IEEE CS Press.

[2] N. Abghour, Y. Deswarte, V. Nicomette, and D. Powell. Specification of authorisation services. Maftia project ist 1999-11583, deliverable d27, January 2001.

[3] N. Abghour, Y. Deswarte, V. Nicomette, and D. Powell. Design of the local reference monitor. Maftia project ist 1999-11583, deliverable d6, April 2002.

[4] A. Adelsbach, D. Alessandri, C. Cachin, S. Creese, Y. Deswarte, K. Kursawe, J. C. Laprie, D. Powell, B. Randell, J. Riordan, P. Ryan, W. Simmonds, R. Stroud, P. Veríssimo, M. Waidner, and A. Wespi. *Conceptual Model and Architecture of MAFTIA. Project MAFTIA deliverable D21*. January 2002. http://www.research.ec.org/maftia/deliverables/D21.pdf.

[5] A. Adelsbach, D. Alessandri, C. Cachin, S. Creese, Y. Deswarte, K. Kursawe, J. C. Laprie, D. Powell, B. Randell, J. Riordan, P. Ryan, W. Simmonds, R. Stroud, P. Veríssimo, M. Waidner, and A. Wespi. *Conceptual Model and Architecture of MAFTIA. Project MAFTIA IST-1999-11583 deliverable D21*. January 2003. http://www.research.ec.org/maftia/deliverables/D21.pdf.

[6] MAFTIA portal: http://http://www.newcastle.research.ec.org/maftia.

[6] A. Avizienis, J.-C. Laprie, and B. Randell. Fundamental concepts of dependability. Research Report 01145 (Revision 1: December 2002), LAAS-CNRS, August 2001. UCLA CSD Report no. 010028; Newcastle University Report no. CS-TR-739.

[7] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, October 1985.

[8] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proc. 9th ACM Conference on Computer and Communications Security (CCS)*, pages 88–97, 2002.

[9] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols (extended abstract). In Joe Kilian, editor, *Advances in Cryptology: CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001.

[10] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 123–132, 2000.

[11] Christian Cachin and Jonathan A. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proc. Intl. Conference on Dependable Systems and Networks (DSN-2002)*, pages 167–176, June 2002.

[12] Ran Canetti, Rosario Gennaro, Amir Herzberg, and Dalit Naor. Proactive security: Long-term protection against break-ins. *RSA Laboratories' CryptoBytes*, 3(1), 1997.

[13] Ran Canetti and Tal Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 42–51, 1993. Updated version available from `http://www.research.ibm.com/security/`.

[14] A. Casimiro, P. Martins, and P. Veríssimo. How to build a Timely Computing Base using Real-Time Linux. In *Proceedings of the IEEE International Workshop on Factory Communication Systems*, pages 127–134, September 2000.

[15] A. Casimiro and P. Veríssimo. Using the Timely Computing Base for dependable QoS adaptation. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, October 2001.

[16] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[17] CIDF. Common intrusion detection framework, 1999.

[18] M. Correia, L. C. Lung, N. F. Neves, and P. Veríssimo. Efficient Byzantine-resilient reliable multicast on a hybrid failure model. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, pages 2–11, October 2002.

[19] M. Correia, P. Veríssimo, and N. F. Neves. The design of a COTS real-time distributed security kernel. In *Proceedings of the Fourth European Dependable Computing Conference*, pages 234–252, October 2002.

[20] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822, 1999.

[21] Y. Deswarte, N. Abghour, V. Nicomette, and D. Powell. An internet authorization scheme using smart card-based security kernels. In I.Attali and T.Jensen, editors, *International Conference on Research in Smart Cards, e-Smart 2001*, Lecture Notes in Computer Science LNCS 2140, pages 71–82, Cannes, France, 2001. Springer-Verlag.

[22] J.E. Dobson and B. Randell. Building reliable secure systems out of unreliable insecure components. In *Conf. on Security and Privacy*, pages 187–193, Oakland, CA, USA, 1986. IEEE CS Press.

[23] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[24] J. Fraga and D. Powell. A fault and intrusion-tolerant file system. In J.B. Grimson and H.-J. Kugler, editors, *IFIP 3rd Int. Conf. on Computer Security*, Computer Security, pages 203–218, Dublin, Ireland, 1985. Elsevier Science Publishers B.V. (North-Holland).

[25] Rachid Guerraoui and Andre Schiper. Transaction model vs. virtual synchrony model: Bridging the gap. In *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*, pages 121–132. Springer-Verlag, 1995.

[26] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Department of Computer Science, May 1994.

[27] L.R. Halme and R.K. Bauer. Aint misbehaving: A taxonomy of anti-intrusion techniques, 2000.

[28] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Proceedings of the 31st Annual Hawaii International Conference on System Sciences*, pages 317–326, January 1998.

[29] C.E. Landwehr, A.R. Bull, J.P. McDermott, and W.S. Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3):211–254, 1994.

[30] J.-C. Laprie, editor. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerance*. Springer-Verlag, Vienna, Austria.

[31] J.-C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *15th IEEE Int. Symp. on Fault Tolerant Computing (FTCS-15)*, pages 2–11, Ann Arbor, MI, USA, 1985. IEEE CS Press.

[32] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[33] F. Meyer and D. Pradhan. Consensus with dual failure modes. In *Proceedings of the 17th IEEE International Symposium on Fault-Tolerant Computing*, pages 214–222, July 1987.

[34] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 707–710, Phoenix, Arizona, April. IEEE.

[35] N. F. Neves and P. Veríssimo, editors. *Complete Specification of APIs and Protocols for the MAFTIA Middleware. Project MAFTIA deliverable D9*. July 2002. http://www.newcastle.research.ec.org/maftia/deliverables/D9.pdf.

[36] V. Nicomette and Y. Deswarte. Symbolic rights and vouchers for access control in distributed object systems. In J. Jaffar and R. H. C. Yap, editors, *Proc. 2nd Asian Computing Science Conference (ASIAN'96)*, LNCS n1179, pages 193–203, Singapore, 1996. Springer-Verlag.

[37] V. Nicomette and Y. Deswarte. An authorization scheme for distributed object systems. In *Proc. Int. Symposium on Security and Privacy*, pages 21–30, Oakland, CA, USA, 1997. IEEE Computer Society Press.

[38] NSA. Nsa glossary of terms used in security and intrusion detection, 1998.

[39] M. Patiño Martínez, R. Jiménez-Peris, and S. Arévalo. Group transactions: An integrated approach to transactions and group communication. In *Workshop on Concurrency in Dependable Computing (at 22nd International Conference on Application and Theory of Petri Nets and 2nd International Conference on Application of Concurrency to System Design*, pages 5–15, Newcastle upon Tyne, UK, 2001.

[40] D. Powell, D. Seaton, G. Bonn, P. Veríssimo, and F. Waeselynk. The Delta-4 approach to dependability in open distributed computing systems. In *Proceedings of the 18th IEEE International Symposium on Fault-Tolerant Computing*, June 1988.

[41] David Powell (Guest Ed.). Group communication. *Communications of the ACM*, 39(4):50–97, April 1996.

[42] Michael O. Rabin. Randomized Byzantine generals. In *Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 403–409, 1983.

[43] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, and A. F. Zorzo. Coordinated atomic actions: from concept to implementation. Technical Report TR 595, Department of Computing, University of Newcastle upon Tyne, 1997.

[44] M. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80, November 1994.

[45] Michael K. Reiter. Distributing trust with the Rampart toolkit. *Communications of the ACM*, 39(4):71–74, April 1996.

[46] Michael K. Reiter and Kenneth P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems*, 16(3):986–1009, May 1994.

[47] Andre Schiper and Michel Raynal. From group communication to transactions in distributed systems. *Commun. ACM*, 39(4):84–87, 1996.

[48] Adi Shamir. How to share a secret. *ACM*, 22(11), November 1979.

[49] Victor Shoup. Practical threshold signatures. In Bart Preneel, editor, *Advances in Cryptology: EUROCRYPT 2000*, volume 1087 of *Lecture Notes in Computer Science*, pages 207–220. Springer, 2000.

[50] Victor Shoup and Rosario Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. In Kaisa Nyberg, editor, *Advances in Cryptology: EUROCRYPT '98*, volume 1403 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1998.

[51] P. Veríssimo and A. Casimiro. The Timely Computing Base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930, August 2002.

[52] P. Veríssimo, A. Casimiro, and C. Fetzer. The Timely Computing Base: Timely actions in the presence of uncertain timeliness. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 533–542, June 2000.

[53] P. Veríssimo, A. Casimiro, and C. Fetzer. The Timely Computing Base: Timely actions in the presence of uncertain timeliness. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 533–542, June 2000.

[54] P. Veríssimo and L. Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, 2001.

[55] P. E. Veríssimo, N. F. Neves, and M. P. Correia. Intrusion-tolerant architectures: Concepts and design. In R. Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[56] Paulo Veríssimo. Uncertainty and predictability: Can they be reconciled? In *Future Directions in Distributed Computing*, pages –. Springer-Verlag LNCS 2584, to appear 2003.

[57] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, page 464. IEEE Computer Society, 2000.

[58] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, page 499. IEEE Computer Society, 1995.

[59] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. Coca: A secure distributed online certification authority. *ACM Trans. Comput. Syst.*, 20(4):329–368, 2002.