Project IST-1999-11583

Malicious- and Accidental-Fault Tolerance for Internet Applications

DESIGN OF THE LOCAL AUTHORIZATION CHECKER

Noreddine Abghour
Yves Deswarte
Vincent Nicomette
David Powell

LAAS-CNRS

MAFTIA deliverable D6

Public document

3 APRIL 2002

Revision History

Rev.	Date	Comments
1.0	01/03/02	Incomplete initial draft.
1.1	05/03/02	Draft.
1.2	12/03/02	Draft
1.3	13/03/02	
1.4	18/03/02	
1.5	26/03/02	
1.6	27/03/02	
2.0	31/03/02	
2.1	2/04/02	
2.2	3/04/02	

Table of Contents

Design of the local authorization checker

Noreddine Abghour
Yves Deswarte
Vincent Nicomette
David Powell
LAAS-CNRS

Abstract: The MAFTIA authorization services implement a fine grain protection, i.e., capable of protecting each object method invocation, in order to satisfy as much as possible the least privilege principle and to provide the most efficient protection. The authorization services are implemented by distributed authorization servers and by local authorization checkers, supported by Java smart cards. This document describes the authorization checker functions and implementation.

1. Introduction

In MAFTIA deliverable D27 "*Specification of Authorization Services*" [Abghour & al. 2001], we have established that protection should be applied at the level of object methods:

- This level enables us to cope with suspicious mobile agents, as well as risks of abuse by local (privileged) users.
- The object model is consistent with the current technology for developing distributed applications in heterogeneous environments (Java, ActiveX, etc.).
- This model is richer and more flexible than the usual client-server model, which is currently used for most interactions on the Internet.

Consequently, the authorization services that we propose for MAFTIA are able to authorize or deny any object method invocation. As explained in [Abghour et al. 2001], the authorization to invoke an object method is represented by a capability issued by authorization servers and verified by local *authorization checkers*[1]. This document describes the functions of the local authorization checkers and their implementation using Java Cards.

---

[1] In [Abghour & al. 2001] the term "*security kernel*" was used for what we call here "*authorization checker*". Here, we reserve the term "security kernel" for that part of the authorization checker that resides on the local Java Card and that has to be tamperproof for the needed security properties to be enforced. This is more consistent with the usual definition of a security kernel.

The document presents first the global architecture of the authorization service in Section 2 and the needed security properties in Section 3. The design of the authorization checker and its implementation are described in Section 4. Section 5 gives an application example.

2. General Authorization Architecture

In [Nicomette & Deswarte 1997], we proposed a generic authorization scheme for distributed object systems. In this scheme, an application can be viewed at two levels of abstraction: *composite operations*[2] 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.

2.1. Authorization proofs

A request from object *O* to run a composite operation is authorized or denied by the authorization server according to symbolic rights stored in an access control matrix. More details on how the authorization server checks if a composite operation request is to be granted or denied are given in [Nicomette & Deswarte 1996] and [Abghour & al. 2001]. If the request is authorized, authorization proofs are created by the authorization server *AS* for all the method executions needed to perform the composite operation. These authorization proofs are delivered in the form of a list of *permissions* for *O* to perform operations op_1, ..., op_n, and the list is signed[3] by the authorization server to prevent forgery:

$$\mathbf{AS}\rightarrow\mathbf{O}:\left\langle\left\{\mathbf{Perm}(O;op_1);\cdots;\mathbf{Perm}(O;op_n)\right\}\right\rangle_{SKas}$$

Two kinds of operations are to be considered: simple method invocations and composite operations. In the first case, the permission contains the identification of the object *O* authorized to invoke the method, the identification of the authorized method *O'.m*, some constraints *parC* on the invocation parameters, the capability for *O* to invoke *O'.m* with these constraints, and optionally a voucher:

$$\mathbf{Perm}(O;O'.m)=\left\{O;O'.m(parC);\mathbf{Cap}(O;O'.m(parC))\left[;\mathbf{Vouch}(O'.m)\right]\right\}$$

The capability **Cap**(*O*;*O'.m*(*parC*)) is a proof that object *O* is authorized to invoke (by a *Rmi*, remote method invocation, in Java) method *m* of object *O'* with some constraints *parC* on the invocation parameters. These constraints can be simply the values of the authorized invocation parameters, or relations on the values of the parameters, such as the maximum authorized value.

---

[2] In previous deliverables, "composite operations" were called "high-level operations".

[3] In our notation, $\langle data\rangle_{PK}$ means that the data are ciphered with the public key *PK*, $\langle data\rangle_{SK}$ means that the data are signed with the private key (or signature key) *SK*, and $\langle data\rangle_{PK,SK}$ means that the data are first ciphered with *PK* then signed with *SK*.

When *O* invokes *O'.m*, the invocation must be accompanied by the capability, which will be checked by the local authorization checker located on the host of *O'*.

If *O'.m* does not make any invocation, $\mathbf{Perm}(O;O'.m)$ does not contain a voucher. But if during its execution, *O'.m* invokes other operations (method executions or composite operations), it will need authorization proofs for these invocations. In that case, $\mathbf{Perm}(O;O'.m)$ contains a voucher, which is a sealed list of permissions for *O'* to invoke op_1, ..., op_n:

$$\mathbf{Vouch}(O'.m)=\langle\{\mathbf{Perm}(O';op_1);\cdots;\mathbf{Perm}(O';op_n)\}\rangle_{SKas}$$

Vouch(*O'.m*) will be transmitted by *O* to *O'* when *O* invokes *O'.m*, in the same way as the list of permissions was transmitted by the authorization server to *O* as a reply to the request by *O* for a composite operation. The voucher has been created by the authorization server during the permission generation, and the voucher is signed by the authorization server to prevent forgery. It must be noted that the authorization proofs represented by the voucher cannot be used directly by *O*. It can only be delegated by *O* to *O'* in order to execute *m* on behalf of *O*.

In the second kind of permissions, i.e., when the operation is a composite operation, the permission contains the identification of the object *O* authorized to invoke the composite operation, the identification of the composite operation *cop*, some constraints *parC* on the invocation parameters, and the token for *O* to invoke *cop* with these constraints:

$$\mathbf{Perm}(O;cop)=\{O;cop(parC);\mathbf{Token}(O;cop(parC))\}$$

Token(*O*;*cop*(*parC*)) is a proof that object *O* is authorized to carry out the composite operation *cop* with some constraints *parC* on the invocation parameters. With this token, object *O* can request from the authorization server all the authorization proofs needed for performing *cop*. In this case, *cop* is a nested composite operation. Thus, the notion of composite operation is recursive: a composite operation is a set of composite operations; a simple method execution is a particular case of a composite operation.

2.2. Discussion

2.2.1. On tokens

Tokens are not strictly necessary in our scheme. Since a token, generated by the authorization server, is an authorization proof to be presented to the authorization server to generate permissions, it would have been possible to generate these permissions in the first place and distribute them instead of the token. But tokens are useful for flexibility and efficiency:

- Flexibility: without tokens, the authorization server, when it receives a request for a composite operation, would have to evaluate the complete invocation tree for all possible executions of the composite operation, and generate all the capabilities for all the possible method invocations. In this case, cycles must be prevented in the invocation sequences. On the contrary, tokens enable dynamic evaluation of authorization at run time.

- Efficiency: without tokens, the authorization server would have to generate all capabilities for all possible method invocations in all possible executions of the composite operation. Most of these capabilities would not be used in the real composite operation execution, so the authorization server would have wasted its time in generating them. Moreover, this means that the objects would receive more capabilities than needed, in contradiction of the least privilege principle. On the contrary, the authorization server will have only to evaluate at run time the tokens corresponding to composite operations that must really executed.

It can also be noted that when an object *O* sends a request to the authorization server for a composite operation *cop*, the reply from the authorization server contains only permissions for *method* invocations (i.e., with capabilities and vouchers). No permissions for composite operations are sent, which means that tokens can only be included in vouchers. Indeed, if a composite operation *cop'* was to be nested in *cop* and if *cop'* was to be directly executed by the same object *O*, it is more efficient to deliver directly to *O* all the capabilities and vouchers needed to execute both *cop* and *cop'*.

2.2.2. On delegation

The voucher-based delegation scheme we have presented above 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 thus *O* could misuse 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 capability to call the spooler and a voucher for the spooler to read the file. The voucher, by itself, cannot be used by *O*. With the capability, *O* can only invoke the spooler and transmit the voucher to the spooler. The spooler can then use capability contained within the voucher to read the file. Thus *O* can have the right to print a file without being authorized to read it.

Let us also note that all the required permissions do not necessarily have to be transmitted in a voucher (or even in a reply to a request for composite operation authorization). The invoked object can have its own rights, stored as symbolic rights in the authorization server access control matrix. Thus, a composite operation *cop* can be launched by *O'* on behalf of *O* (i.e., by running a method *O'.m* invoked by *O*) without *O* delivering a voucher for *cop*, if *O'* is authorized to perform *cop* by itself.

2.2.3. On transient objects

Only persistent objects are known by the authorization server AS. Transient objects, such as a buffer or a temporary file, created for the execution of a single composite operation are not known by AS, and thus AS cannot generate capabilities for such objects. The capabilities for transient objects are completely managed by the authorization checker associated with that host

(a transient object capability can be ciphered by a persistent symmetric key of the local authorization checker; this symmetric key is never transmitted out of the authorization checker). A transient object can be created dynamically by another object, which then receives from the authorization checker all capabilities for this object and it can transmit these capabilities to other local objects at method invocation. Transient object capabilities cannot be transmitted to remote objects.

2.3. Authorization server

Since only composite operations 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 composite operations, they do not have to consider all the elementary rights to invoke object methods. Moreover, since only one request has to be checked by the authorization server for each composite operation, the communication overhead can be reduced.

The authorization server is a trusted third party (TTP), which could be a single point of failure, both in case of accidental failure, or in case of successful intrusion (including by a malicious administrator). To prevent this, in the MAFTIA authorization architecture [Abghour & al. 2001], the authorization server will be made tolerant to accidental faults and intrusions: an authorization server is made of diverse security sites, operated by independent persons, so that accidental faults and malicious intrusions can be tolerated without degrading the service, as long as only few security sites are affected. The global architecture is given by Figure 1

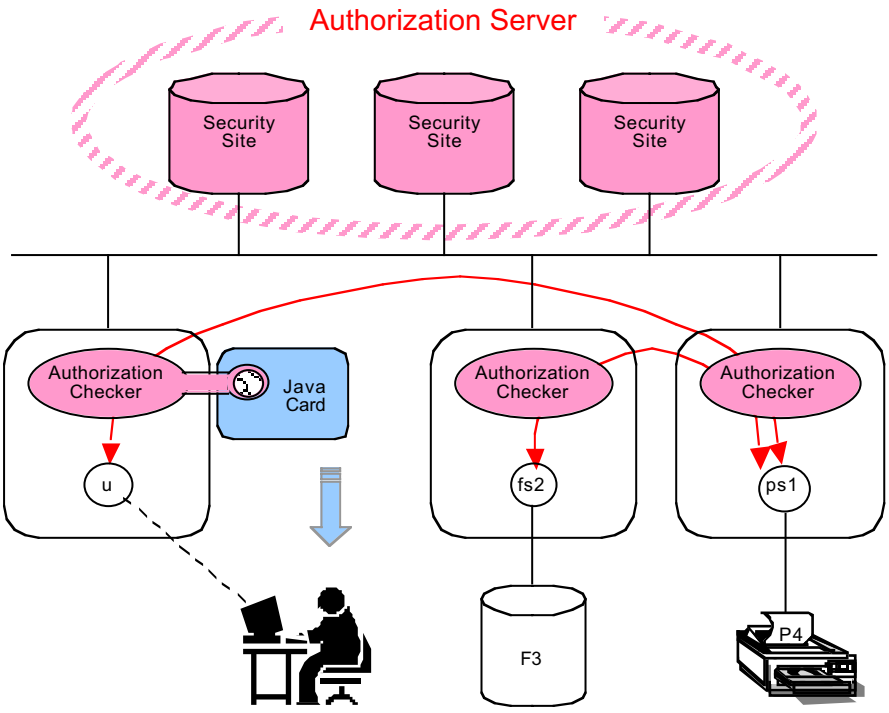

Fig.1. Authorization Architecture

In order to tolerate the failure of one or a small number of the sites composing the authorization server, several protocols specified in D23 [Cachin & al 2001] and D26 [Algesheimer & al 2001], are used:

- *Atomic multicast transmission*: the request sent by an object to the authorization server is distributed to all the sites composing the authorization server. The multicast communication is reliable in the sense that all requests (sent by a non-faulty site) are received correctly by all non-faulty receiving sites and atomic in the sense that non-faulty receiving sites receive identical message sets in the same order. This condition is useful for achieving mutual agreement.

- *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[4].

- *Threshold signature*: the capabilities, tokens 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 token, 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*).

The dialogue between a MAFTIA object and the authorization server is typically as follows.

Object *O* asks the authorization server for the authorization to carry out a composite operation in the system. If the authorization is granted by a majority of the sites composing the authorization server, the authorization server returns to *O* a set of permissions for method invocations:

$$AS \rightarrow O : \left\langle \{ \mathbf{Perm}(O;O_1.m_1); \cdots; \mathbf{Perm}(O;O_n.m_n) \} \right\rangle_{SKas}$$

Each of these permissions thus contains a capability $\mathbf{Cap}(O;O_i.m_i\ (parC))$

To prevent capability forgery or abuse, each capability is ciphered with the public key $PK_{\mathrm{h}(Oi)}$ of the host where O_i is located, and signed by the global private key SK_{as} of the authorization server AS[5]:

$$\mathbf{Cap}\big(O;O_i.m_i(parC)\big) = \left\langle O;O_i.m_i;parC;nonce \right\rangle_{PK_{h(Oi)},SK_{as}}$$

---

[4] 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.

[5] Let us note that in this document, capabilities are the only authorization proof that need to be ciphered and signed; all the others are only signed, because the knowledge of their content cannot give any authorization privilege. Nevertheless, it could also be useful to cipher (with the authorization checker's public key) the lists of permissions and the vouchers, if the knowledge of their content is considered as a breach of privacy.

The nonce is used to prove the freshness of the capability and prevent replay.

If the capability is accompanied by a voucher, this voucher contains a set of permissions needed by O_i to execute m_i on behalf of O. The voucher itself is signed by the global private key SK_{as} of the authorization server:

$$\mathbf{Vouch}(O_i.m_i)=\langle\{\mathbf{Perm}(O';op_1);\cdots;\mathbf{Perm}(O';op_n)\}\rangle_{SKas}$$

This set of permissions can contain method permissions and composite operation permissions. The processing of the method permission is the same as above. Each composite operation permission contains a token **Token**(*O'*;*cop*(*parC*)) which is signed by SK_{as}, the private key of the same authorization server. Indeed, the token is only used by the authorization server as a proof that the composite operation is authorized. So, it has to be signed (with a nonce) to prevent forgery, but there is no need to cipher it since no other host can use it:

$$\mathbf{Token}(O';cop(parC))=\langle O';cop;parC;nonce\rangle_{SKas}$$

2.4. Authorization checker

There is an authorization checker on each host participating in a MAFTIA-compliant application. The authorization checker is responsible for granting or denying local object method invocations, according to capabilities generated by the authorization server or by the authorization checker itself, for transient local objects). In the context of wide-area networks (such as the Internet), the implementation of such an authorization checker is complicated since, due to the heterogeneity of connected hosts, it would be necessary to develop one version of the authorization checker 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 authorization checker, or the same version thereof. This is why we have chosen to implement them using Java Cards.

A distributed MAFTIA application is executed by means of method invocations between MAFTIA objects located on MAFTIA-compliant hosts. An authorization checker in the context of a MAFTIA host is composed of two parts: a Java Card-resident security kernel and a host-resident dispatcher. The dispatcher is a local object, run by each MAFTIA host, that is in charge of controlling all invocations by/to local objects. In fact, the dispatcher also stores and manages the permissions that have been granted to the host's objects: in order to improve the application transparency, the permissions are not delivered to objects, they are stored and used by the dispatcher. For instance, when an object invokes another object, the dispatcher checks the permissions belonging to the invoking object to verify it has indeed a capability to realize this invocation and then inserts the capability and (generally) the voucher into the invocation message and sends the whole message to the destination dispatcher, which will then check the capability and store the permissions included in the voucher for the invoked object.

The security kernel has the responsibility of deciding whether or not to authorize the invocation of particular methods on particular objects on the local host by checking that the corresponding capabilities are presented. These checks represent the central part of the authorization scheme, and thus have to be protected as strongly as possible. We have chosen to implement them on a Java Card, which we consider to be sufficiently tamper-resistant. Note 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 Internet, any hacker can easily have these privileges on his own computer! With capability checks run on the Java Card, we can be confident 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 the host for which the capability is valid. The hacker's privilege on his host gives him no privilege outside that host. The next section details the security properties provided by the different components of the authorization architecture.

3. Security properties

The security of the whole system is ensured by the collaboration between the authorization server and the two components of the local authorization checkers: the card-resident security kernel and the host-resident dispatcher. We present in this chapter the security properties required for these three entities.

3.1. Security properties of the authorization server

As we previously mentioned in Section 2, the authorization server is in charge of creating all the authorization proofs that allow objects in the system to execute operations. As a consequence, the failure of the authorization server would compromise the security of the entire system (i.e., the security of all the applications of the system). Thus, the following security properties need to be enforced by the authorization server:

AS1: The authorization server generates only valid proofs of authorization.

AS2: It is not possible to prevent the authorization server from generating valid proofs of authorization.

To implement these properties, the authorization server must thus be made tolerant to accidental faults and malicious intrusions. To achieve this, the authorization server is composed of several security sites, so that the failure (whether due to an accidental fault or an intrusion) of a small number of security sites can be tolerated without degrading the service globally provided by the authorization server. To this end, two types of mechanisms are used:

- Consensus, so that the non-faulty[6] sites take together one decision.
- Threshold cryptographic techniques for the set of security sites to generate common authorization proofs (the security sites must together generate correctly signed authorization proofs, even if a small number of them are faulty).

Regarding property AS2, we assume that fault tolerance mechanisms are implemented in the network so that no network denial of service attack can succeed and no network failure can occur.

---

[6] By non-faulty, we mean a site that is not compromised by malicious attacks or affected by accidental faults.

3.2. Security properties of the security kernel

Our authorization schemes are intended for Internet applications, i.e., applications that require the cooperation of several geographically distributed objects that distrust one another. In that sense, we cannot trust the hosts where these different objects execute their part of the application. Each host is locally operated by a person who is not necessarily trustworthy. This operator does not trust the other operators of the other MAFTIA hosts and we consider that any operator has enough control to do anything on his local site.

Since any MAFTIA host may be faulty, it must include trusted security functions that constitute the "security kernel" of the host. These security functions allow the permissions that are received locally to be checked. These permissions generated by the authorization server, allow some objects to invoke operations on local objects.

The security kernel must be designed and implemented in such a way that:

SK1: Only authorized operations will be executed on a non-faulty host.

SK2: It is not possible to prevent the execution of an authorized operation on a non-faulty host.

The second property SK2 means that a faulty host cannot provoke the execution of unauthorized operations on a non-faulty host and that a faulty host cannot impersonate a non-faulty host without being detected.

To implement these properties, we choose to use cryptographic keys and algorithms embedded in a Java Card. These cryptographic means constitute "the security kernel". A private/public key pair, associated to each MAFTIA host is generated and stored on the Java Card associated to this host. The public key of the authorization server is also stored on each Java Card (the public keys are stored on the Java Card for integrity reasons, to prevent attacks that would modify of these keys). Each capability, generated by the authorization server, is encrypted by the public key of the MAFTIA host where the invoked object resides and then signed with the private key of the authorization server. These techniques allow the property SK1 to be implemented: on a non-faulty host, if the capability is not correctly encrypted and signed, the corresponding operation will not be executed[7]. We trust the security kernel to correctly check the capabilities. We think it reasonable to trust the security kernel because we consider that modifying the Java code embedded in the Java Card is very difficult and because we consider that obtaining the cryptographic keys that are stored on the Java Card is very difficult. Furthermore, a faulty host cannot generate false permissions for other MAFTIA hosts because these permissions must be signed by the private key of the authorization server.

Regarding SK2, as we want to prevent a faulty host to impersonate a non-faulty host without being detected, we have implemented an acknowledgement mechanism: each time an authorization proof is received by a MAFTIA host, the latter returns an acknowledgement

---

[7] The same is true for transient local objects, except that the capabilities are generated and verified by the same local authorization checker. The transient object capabilities are ciphered — with a secret symmetric key —by the security kernel. The symmetric key is maintained by the Java Card security kernel.

signed with its private key to the requester. This acknowledgment is accompanied by a certificate generated by the authorization server. Even if a faulty host tries to impersonate a non-faulty host by intercepting all the messages for this non-faulty host, it cannot send this acknowledgement because it cannot encrypt it with the private key of the non-faulty host (this key can only be read by the security kernel of the non-faulty host, i.e., within the Java Card; this information is never transmitted outside of the Java Card). As a consequence, even if this faulty host intercepts messages for the non-faulty host in such a way that the non-faulty host no longer receives them (denial of service attack), the attack will be detected because the requester will not receive correct acknowledgements. This protection mechanism is also useful for network denial of service attacks or network failures. We can rely on this acknowledgment mechanism because we trust each security kernel not to reveal the cryptographic keys. We trust it not to reveal the keys because we consider that accessing data stored on the Java Card is sufficiently difficult.

Let us put the emphasis on the following point: the mechanisms presented are not sufficient to guarantee the correct behaviour of a whole MAFTIA application. If a MAFTIA host is corrupted, the local processes can do anything locally, and thus can totally change their behaviour with respect to the whole application. They can decide to invoke operations on local objects that do not form part of the composite operation, or invoke constantly the same function of the same local object, or not invoke correct and authorized operations on objects which located on other hosts. The mechanisms we implement aim to enforce the security policy of the system and cannot guarantee that the behaviour of each process is correct as regards the whole application it is part of. Other mechanisms are included in the MAFTIA architecture to deal with these problems (replication of applications, consensus mechanisms).

In summary, the authorization mechanisms guarantee that only authorized operations will be executed on a non-faulty host and guarantee that a faulty host cannot prevent the other non-faulty hosts from operating correctly.

3.3. Security properties of the dispatcher

The dispatcher object must be present on each MAFTIA host and is in charge of intercepting all the invocations to local objects to check if the access is authorized. To do so, the dispatcher checks the capabilities carried by the invocation message by using the cryptographic algorithms and keys that are stored on the Java Card (the security kernel). In fact, the dispatcher cannot directly read the private key that is stored on the Java Card. It simply asks the security kernel (in the Java Card) to check the signatures of the received capabilities and to decipher the capabilities.

It is important to note that we do not make any particular assumption on the security of the dispatcher object. We do not need particular mechanisms to protect it. Like the JVM, the operating system or the other objects of the host, it may be faulty and stop executing its tasks correctly. Even if it is faulty, we are sure that this object cannot provoke the execution of unauthorized operations *outside* of its host. Since the local Java Card is trusted, this object cannot directly read the private key stored in the Java Card. So, a corrupted dispatcher cannot by itself check the capabilties and furthermore, it does not have the possibility to create fake

permissions. In that sense, it cannot provoke the execution of unauthorized operations on other hosts.

4. Authorization checker implementation

To enforce the security properties specified in the previous chapter, we detail here after the architecture of the authorization checker and the Java Card implementation of the security kernel.

4.1. Java Card overview

First, let us give a short introduction to Java Card technology [Chen 2000]. A Java Card provides means of programming smart cards with a subset of the Java programming language. Smart cards communicate with the rest of the world through Application Protocol Data Units (APDUs, ISO 7816-4 standard). The communication is done in master-slave mode. The master/terminal application initializes the communication by sending a command APDU to the card and the card replies by sending a response APDU. In the case of Java smart cards, besides the operating system, the card's ROM contains a Java Card Virtual Machine (JCVM) that implements a subset of the Java programming language and allows Java Card applets to be executed on the card. The Java Card also contains the standard Java Card API, which provides support for handling APDUs, Application Identifiers (AIDs), Java Card specific system routines, PIN codes, etc. Each Java Card applet is a subclass of the javacard.framework.Applet class and should implement the install method responsible for the initialization of the applet and a process method for handling incoming command APDUs and sending the response APDUs back to the host. Several applets can exist on a single Java Card, but only one can be active at a time.

4.2. The development process

Developing applets for Java Cards involves a procedure somewhat different from that of developing a standard Java application. The Gemplus development environment, called GemXpresso RAD[8], provides tools that simplify the development process. There are several components that make up GemXpresso software, including the Java Card converter tool, a terminal upload tool, and an installation program.

Development of a Java Card applet begins as with any other Java program: after writing one or more Java classes, the source code is compiled with a Java compiler, producing one or more class files. Then, when an applet is to be downloaded to a Java Card, the class files composing the applet are converted to a CAP (Converted Applet) file using a Java Card Converter. The Java Card Converter takes as input not only the class files to be converted, but also one or more

---

[8] GemXpresso RAD 211 is a fully integrated suite of Java Card development tools. It provides all the tools needed to ensure a complete development process in a Java Card environment. It contains several software components to develop, load, debug and test off-card applications and Java Card applets.

export files. When an applet is converted, the converter can also produce an export file for that package. After conversion, the installation tool on the terminal loads the CAP file and transmits it to the Java Card. An installation program on the card receives the contents of the CAP file and prepares the applet to be run by the Java Card Virtual Machine.

4.3. The authorization checker architecture

An authorization checker in the context of a MAFTIA host is composed of a Java Card and a dispatcher. The dispatcher is a local object that always runs on each MAFTIA host and that is in charge of dispatching remote requests towards the local objects. The dispatcher is an object run by the host Java Virtual Machine (JVM), which may be the JVM of a browser such as Netscape or Internet Explorer. This JVM must allow the use of digital signatures as a mechanism for verifying that an object code to be loaded comes from a trusted source. The current Java security mechanism is flexible enough to allow the addition of digitally signed applets. The *ClassLoader* class first checks the digital signature of the loaded code using the MAFTIA public key PK_m, to determine whether the class loading should be allowed or denied.

The local authorization checker enforces access control via the dispatcher. The latter is considered as the host-resident part of the authorization checker (see Fig.2). It manages the complete runtime environment for application objects and is the security controller on a MAFTIA compliant host. The dispatcher is invoked each time a local or remote object attempts to perform an invocation of a local object method (*capability verification*) and when a local object performs an invocation of a remote object method (verification that the local object owns a permission for this invocation). The capability verification is the core of our authorization scheme. In our design, this verification relies on strong cryptographic algorithms run on a Java Card. Indeed, if this verification had been run on the host itself, it could have been easily hacked by an operator having a complete control of the host, or even by a remote intruder gaining the same privileges. On the contrary, being run on a tamper-resistant smart card, this verification code can be considered as trustworthy enough to enforce the security properties specified in Section 3.2. This is why the capability verification is delegated by the dispatcher to the security kernel.

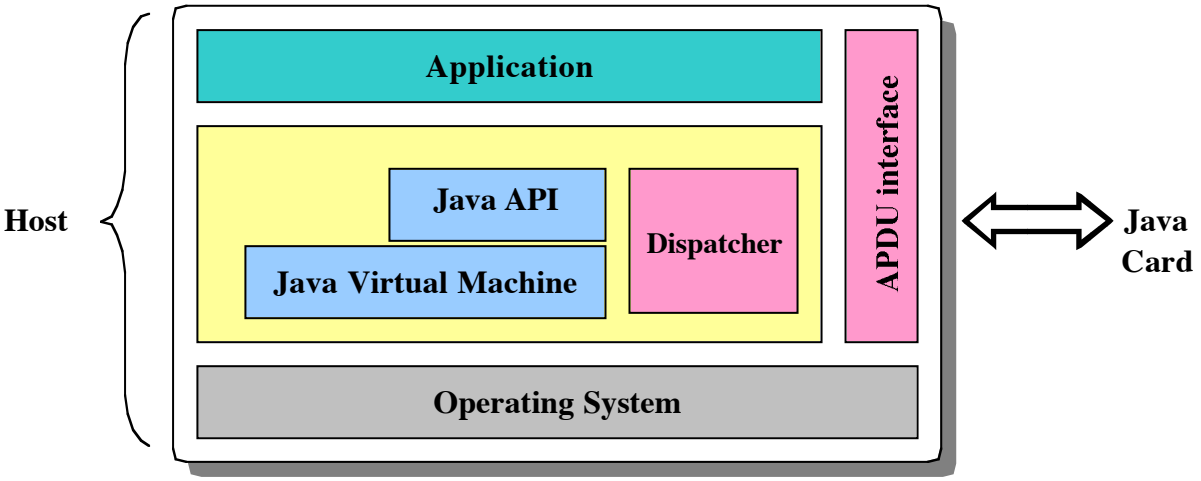

Fig.2. Architecture and environment of the host-resident part of the authorization checker

The security kernel is the Java Card-resident part of the authorization checker. It comprises a program code launched when the smart card is inserted. This code has the responsibility of verifying the validity of a capability for a given invocation. The verification procedure is described below in Section 4.5.

Interactions between the dispatcher and the security kernel are run by exchanging pairs of APDUs (Application Data Units) through the smart card reader. These interactions are initiated by the dispatcher: the dispatcher sends a *CommandAPDU* to the Java Card. In return, the card sends back a *ResponseAPDU* to the dispatcher.

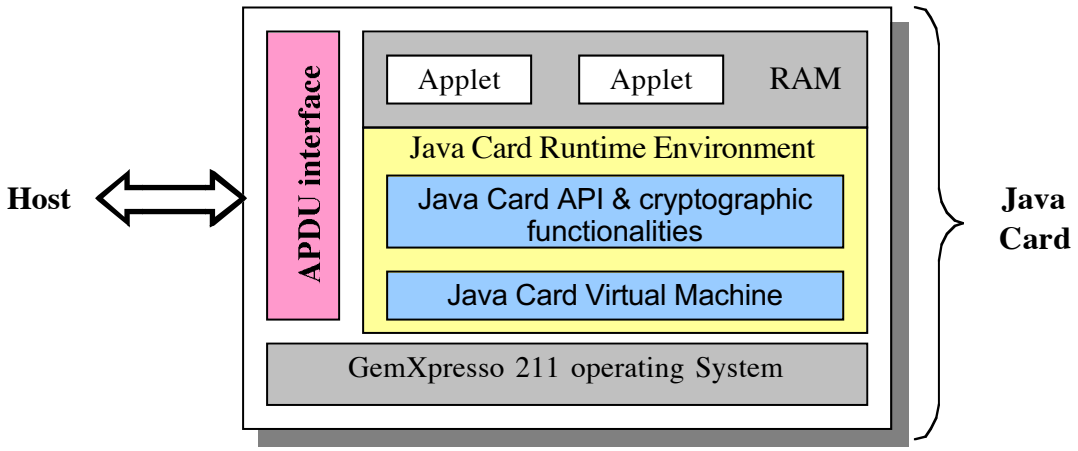

Fig.3. Architecture of the card-resident part of the authorization checker

4.4. Cryptographic keys

Our implementation is based on the GemXpresso 211Pk card, which offers 20 to 32kbit of EEPROM memory, features both the traditional DES (Data Encryption Standard) crypto system and an RSA (Rivest Shamir and Adelman) algorithm for public key crypto. Key calculation software and a dedicated co-processor have been integrated specifically for the RSA algorithm, which requires many multiplications of large integer numbers and consequently requires significant processing power.

In our case, several cryptographic keys must be stored in the Java Card:

- The MAFTIA public key, that we note PK_m. This key is associated with the MAFTIA private key SK_m, which is not stored in the Java Card. SK_m is used to sign MAFTIA compliant code.
- A private/public key pair specific to Java Card j, that we note SK_j, PK_j.
- The authorization server public key, PK_{as} (this key is associated with the private key shares of the authorization sites, using by a threshold signature algorithm, see Section 2.3).
- The secret symmetric key K_i used to encrypt the capabilities for local transient objects.

Other information can be stored in the Java Card, for instance for the authentication of the user. If the host is a personal workstation, the Java Card belongs to the user, and the private/public key pair stored on the card is specific to the user. The authentication information can be a

certificate, which associates the user public key PK_j with some attributes (e.g., the user name, his function or title, his date of birth, etc.), collectively signed by a certification authority. The following authentication protocol can be run by the Java Card:

- The user inserts the Java Card in the reader and enters his PIN (on the host keyboard or on the card reader keyboard, depending on the implementation) to activate the card.
- The user Java Card sends the certificate to the authenticator (the local host or a remote host).
- The authenticator replies by a challenge (e.g., a random number) ciphered with the public key included in the certificate.
- The Java Card deciphers the challenge by using the private key, computes a response (e.g., challenge plus one) and sends it back to the authenticator.
- The authenticator compares the response with the result computed locally.

If the host is a server, it can be administrated by several persons, each one being provided with a Java Card for the server. In this case, all the server administrator Java Cards must share the same pair SK_j, PK_j. If the above authentication protocol is used, each administrator needs to have a personal certificate, so the personal private/public key pair SK_a/PK_a of the administrator must be different from the server private/public key pair SK_j/PK_j.

The next section details the Java Card implementation of the security kernel.

4.5. Capability verification

A distributed application in the context of MAFTIA is executed by means of method invocations between MAFTIA objects located on MAFTIA-compliant hosts. The MAFTIA object codes are signed off-line by the global MAFTIA private key SK_m, and this signature is checked by the local JVM of the MAFTIA host by using the corresponding public key PK_m, which can be read on the Java Card by the JVM. The invocation between objects is carried out through Java RMIs (Remote Method Invocations). The invocation parameters and returned values are thus exchanged between objects by passing messages corresponding to these Java RMIs.

Each capability is ciphered by the public key PK_j of the Java Card of the host where the invoked object is located and signed by the private key SK_{as} of the authorization server. So the capability verification is done in three steps: first the security kernel checks the signature of the capability by using the authorization server public key PK_{as}; second, the security kernel deciphers the capability with its own private key SK_j; finally, the security kernel verifies that the invocation is compatible with the capability content (identity of the invoker, identity of the invoked method including the invoked object identity, compatibility of the invocation parameters with the parameter constraints included in the capability).

As been explained in Section 2, a capability is generally accompanied by a voucher which contains the permissions needed for the invoked method to be authorized to invoke other operations.

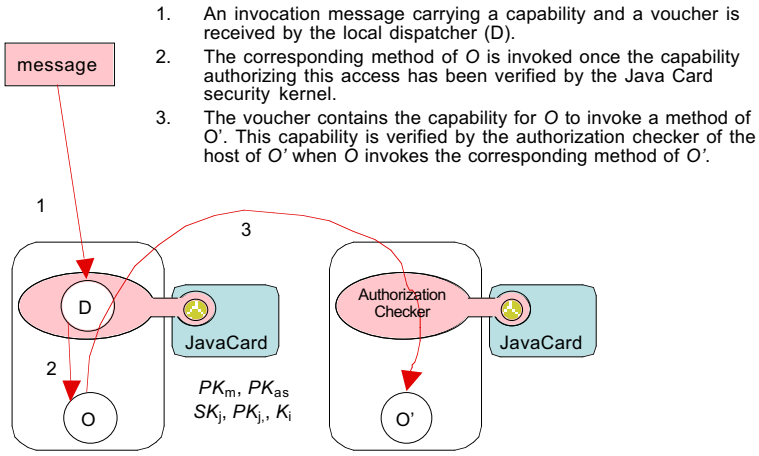

Fig.4. Example of a voucher containing a capability on a remote object.

A permission can be a capability to invoke an object method or a token to initiate a composite operation. If the permission is a capability, this capability is ciphered by the public key of the Java Card associated with the host on which the invoked object resides and will be checked by the corresponding security kernel (step 3 of Fig.4). If the permission is an authorization to carry out a composite operation, the token is presented to the authorization server as a proof that the object is authorized to carry out that operation (step 3 of Fig.5). The authorization server simply checks the validity of the token and then delivers to the object the corresponding permissions.

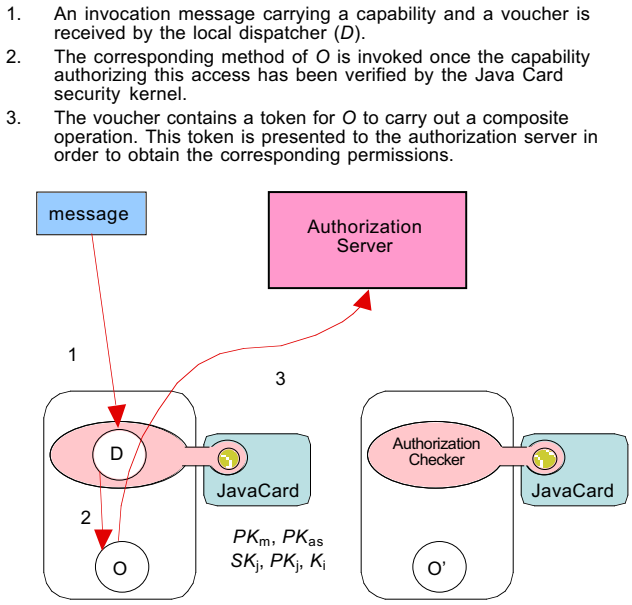

Fig.5. Example of a voucher containing a token.

5. Application example

In the following example, we consider that a doctor wants to send a copy of a patient medical file to another healthcare professional. The set of objects that take part in the execution of this composite operation are presented in Figure 6. In this figure, *U* is the doctor, *V* is another healthcare professional (with role *HCP*)[9], *DBS* (of class *DATABASESERVER*) is a database server, Pmf_1 (of class *PATIENTMEDICALFILE*) is the medical file of a patient of Doctor *U*. *MTA*1 and *MTA*2 are mail transport agents, which are in charge of transmitting electronic mail. The class of these objects is *MTA*. Finally, *tf* is a transient file located on the host of *DBS*.

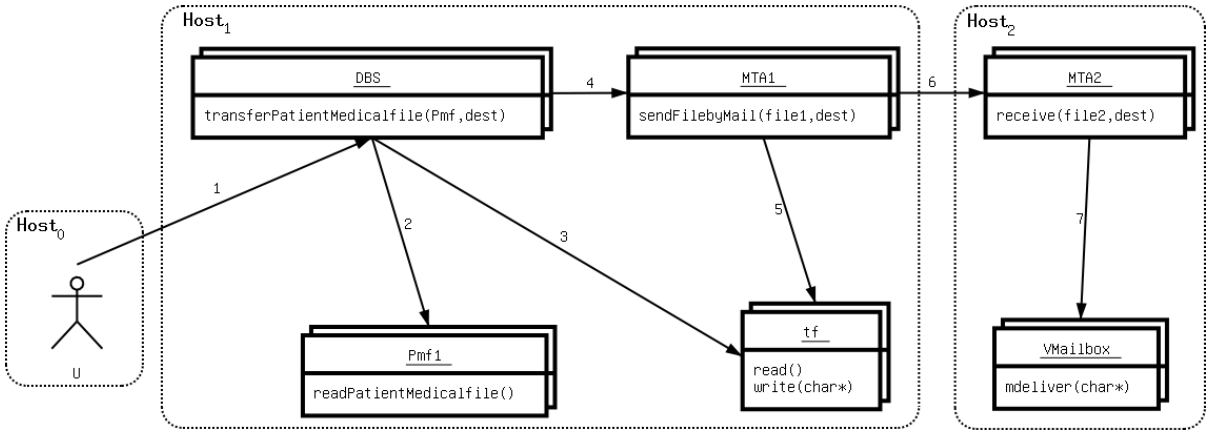

Fig.6. Example of a composite operation

In the access control matrix, *U*'s row holds the symbolic right to carry out the composite operation *SendPatientMedicalfile* denoted by *SPmf*. Detailed descriptions of symbolic right are specified in D27 [Abghour & al. 2001]. Using delegation of rights through vouchers, a sufficient access control matrix is given in table 1.

	…	*HCP Role*	*{Pmf(U)}*
U		$SPmf(\{Pmf(U)\}, this)$	*read, write(*),* $SPmf(this, HCPRole)$
DBS	…		
…			

Table.1. Access control matrix

Let us denote by D_i the dispatcher running on the $Host_i$ and by JC_i the JavaCard connected to the $Host_i$. In the following, the invocations presented in Figure 6 are denoted by $\overset{i}{\mapsto}$.

[9] We also use the notation *U* and *V* to designate the objects representing these persons in the information system

The scenario that enables *U* to transfer a copy of a patient medical file to another healthcare professional is described here. We indicate in this scenario the functions of the authorization server and the authorization checkers with respect to capability distribution and verification.

Step 1: $\boldsymbol{U \rightarrow AS}$: Request$\left(SendPatientMedicalFile\left(Pmf_1,V\right)\right)$

User *U* located on the $Host_0$ wants to send a patient medical file to a user *V* located on the $Host_2$. The user *U* asks the authorization server for the authorization to execute the composite operation $SendPatientMedicalfile\left(Pmfi,V\right)$. The authorization server checks that in the access matrix *U*'s row holds sufficient symbolic rights to grant the authorization of this operation. In this case, *U* has the symbolic right *SPmf* to send a patient medical file to any user with the role *HCP* (HealthCare Professional), and for the *Pmf* of all *U*'s patients[10]. Since *V* is a healthcare professional, $SPmf\left(Pmfi,HCP\right)$ and $SPmf\left(PatientMedicalfile,V\right)$ are both authorized and the composite operation authorization has to be granted.

Consequently, the authorization server replies by a list of permissions signed by the global private key SK_{as} of the authorization server. The list contains only one permission for *U* to invoke one method of the object *DBS*:

$$\boldsymbol{AS \rightarrow D_0} : \left\{ \begin{array}{c} U;DBS.transferPatientMedicalfile(Pmf_1,V);\mathbf{Cap}\left(U;DBS.transferPatientMedicalfile(Pmf_1,V)\right); \\ \mathbf{Vouch}\left(DBS.transferPatientMedicalfile\right) \end{array} \right\}$$

$\mathbf{Cap}\left(U;DBS.transferPatientMedicalfile(Pmf_1)\right)$ is the capability[11] for *U* to invoke *DBS*'s method *transferPatientMedicalfile*, with a constraint on the invocation parameters: this capability is valid only for the patient medical file Pmf_1 and the destination *V*. This capability is encrypted by the public key of the host where *DBS* is located and signed by the authorization server's private key.

The voucher $\mathbf{Vouch}\left(DBS.transferPatientMedicalfile\right)$ represents the set of permissions needed by the method *DBS.transferPatientMedicalfile* for its execution. Its signed by the authorization server's private key.

Step 2: U invokes the *transfertMedicalPatientfile* method of *DBS* with parameters corresponding to the specific patient medical file Pmf_1 and to the healthcare professional *V* (invocation 1 of Fig.6). D_0 checks that it owns a capability for this invocation (received at Step 1) and then inserts into the invocation message the capability and the corresponding voucher:

$$\boldsymbol{U} \overset{1}{\mapsto} \boldsymbol{DBS} : \left\{ \begin{array}{c} \mathbf{RMI}\left(DBS.transferPatientMedicalfile\left(Pmf_1,V\right)\right); \\ \mathbf{Cap}\left(U;DBS.transferPatientMedicalfile(Pmf_1,V)\right);\mathbf{Vouch}\left(DBS.transferPatientMedicalfile\right) \end{array} \right\}$$

The dispatcher D_1 running on $Host_1$ intercepts this invocation and invokes its security kernel to verify the capability:

---

[10] {*Pmf(U)*} represents the set of the medical files of the patients of Doctor *U*. In this access control matrix, *U* also owns the rights to invoke methods read and write of any object in the set {*Pmf(U)*}.

[11] **Cap**(*O*;*O'.m*) denotes the capability for object *O* to invoke method m of object *O'*. **Vouch**(*O'.m*) represents a set of permissions for *O*' delegated by the object invoking method m of O'.

$$\boldsymbol{D_1 \rightarrow JC_1}: \left\{ \begin{array}{c} \mathbf{RMI}\left(DBS.transferPatientMedicalfile\left(Pmf_1,V\right)\right); \\ \mathbf{Cap}\left(U;DBS.transferPatientMedicalfile(Pmf_1,V)\right);\mathbf{Vouch}\left(DBS.transferPatientMedicalfile\right) \end{array} \right\}$$

The Java Card security kernel first verifies the signature of the capability, using PK_{as}. Then it deciphers the capability using SK_1, and returns its content to the dispatcher. Finally the dispatcher verifies that the invocation is compatible with the capability content (invoker = *U*; invoked method = *DBS.transferPatientMedicalfile*; parameters = Pmf_1, *V*).

When the capability is verified, the security kernel verifies the signature of the voucher using PK_{as} and generates an acknowledgement (signed by SK_1), which is returned to the dispatcher D_1. The dispatcher returns the acknowledgement to D_0, invokes the method *DBS.transferPatientMedicalfile*(Pmf_1,*V*), and stores the two permissions included in the voucher for *DBS*, one for invoking the method *readPatientMedicalfile* of object Pmf_1 (with no parameter), and another to invoke method *sendFilebyMail* of object *MTA*1 with no constraint on the first parameter (the absence of constraints is noted "*"), and with the constraint that the second parameter must be equal to *V* (the latter capability is accompanied by a voucher):

$$\left\{ \begin{array}{c} DBS;Pmf_1.readPatientMedicalfile;\mathbf{Cap}\left(DBS;Pmf_1.readPatientMedicalfile\right); \\ DBS;MTA1.sendFilebyMail(*,V);\mathbf{Cap}\left(DBS;MTA1.sendFilebyMail(*,V)\right); \\ \mathbf{Vouch}\left(MTA1.sendFilebyMail\right) \end{array} \right\}$$

Step 3: *DBS* invokes method *readPatientMedicalfile* (invocation 2 of Fig.6) to read the patient medical file Pmf_1:

$$\boldsymbol{DBS} \overset{2}{\mapsto} \boldsymbol{Pmf_1}: \left\{\mathbf{Rmi}\left(Pmf_1.readPatientMedicalfile(\ )\right); \mathbf{Cap}\left(DBS;Pmf_1.readPatientMedicalfile\right)\right\}$$

D_1 sends the invocation message to the security kernel located on Pmf_1*'s* host (i.e., the same Host_1). The security kernel (located on JC_1) checks the validity of the capability $\mathbf{Cap}\left(DBS;Pmf_1.readPatientMedicalfile(\ )\right)$ and generates an acknowledgement to D_1. *DBS* receives data from the patient medical file and creates a temporary file *tf* to be used by *MTA1*. When creating *tf*, *DBS* receives the owner capability on this file from its local authorization checker. Then *DBS* copies Pmf_1 into *tf* by the way of invocation 3 of Fig.6 (the write access to *tf* is authorized by D_1 because *DBS* possesses the owner capability on *tf*).

$$\boldsymbol{DBS} \overset{3}{\mapsto} \boldsymbol{tf}: \left\{\mathbf{RMI}\left(tf.write(\langle \text{contentof}\, Pmf1 \rangle)\right)\right\}$$

DBS asks the authorization checker to create capabilities for *MTA1* to access *tf* methods read and delete (the dispatcher authorizes this capability creation because *DBS* possesses the owner capability on *tf*).

Step 4: *DBS* invokes method *sendFilebyMail* of *MTA1* (invocation 4 of Fig.6) and transfers *tf* read and delete capabilities and the voucher $\mathbf{Vouch}\left(MTA1.sendFilebyMail\right)$ to *MTA1*.

$$\boldsymbol{DBS} \overset{4}{\mapsto} \boldsymbol{MTA1}: \left\{ \begin{array}{c} \mathbf{RMI}\left(DBS.sendFilebyMail\left(tf,V\right)\right); \\ \mathbf{Cap}\left(DBS;MTA1.sendFilebyMail(*,V)\right);\mathbf{Vouch}\left(MTA1.sendFilebyMail\right) \end{array} \right\}$$

The dispatcher located on the host of *MTA*1 (i.e., D_1) intercepts this invocation, retrieve the corresponding capability and voucher and transfers them to the local Java Card JC_1.

The on-card authorization checker located on JC_1 verifies the signature and decipher the capability for the dispatcher to verify the capability validity to invoke the method *sendFilebyMail* of *MTA*1. Next JC_1 verifies the signature of the voucher $\mathbf{Vouch}(MTA1.sendFilebyMail)$, then it returns an acknowledgment to D_1.

D_1 then invokes *MTA1.sendFilebyMail* and stores the content of the voucher, which is a token: $\mathbf{Token}(MTA1;DeliverFilebyMail(*,V))$ for *MTA1* to perform the composite operation *DeliverFilebyMail* with no constraint on the first parameter and the value *V* for the second.

Step 5: *MTA1* asks the authorization server for the authorization to execute the composite operation $DeliverFilebyMail(*,V)$, this request is intercepted by D_1 which inserts the token in the request message.

$$\boldsymbol{MTA1 \rightarrow AS:}\ \text{Request}(DeliverFilebyMail(*,V));\mathbf{Token}(MTA1;DeliverFilebyMail(*,V))$$

The authorization server checks that the token presented by *MTA1* is valid (i.e., is a right for *MTA1* created by the authorization server). Then the authorization server identifies *MTA2* as the mail transport agent for *V*, and finally gives *MTA1* the permission corresponding to the action $DeliverFilebyMail(*,V)$, which is a capability for *MTA1* to invoke *MTA2.receive* (with the same constraints on the parameters), accompanied by a voucher:

$$\boldsymbol{AS \rightarrow D_1}:\{MTA1;MTA2.receive(*,V);\mathbf{Cap}(MTA1;MTA2.receive(*,V));\mathbf{Vouch}(MTA2.receive)\}$$

This permission is signed by the private key SK_{as} of the authorization server.

$\mathbf{Vouch}(MTA2.receive)$, is a voucher that has to be delegated to *MTA2* to perform the next part of the composite operation.

MTA1 calls method read of *tf* (invocation 5 of Fig.6). This invocation is controlled by *tf's* local authorization checker (i.e., D_1), which verifies that *MTA1* holds the capability for method read of *tf*.

Step 6: Using the capability $\mathbf{Cap}(MTA1;MTA2.receive(*,V))$, *MTA1* invokes *MTA2*'s method $receive(\langle Content\ of\ tf\rangle,V)$ (message 6 of Fig.6). Once this method has been invoked, *MTA1* sends a request to delete *tf* (the operation is authorized since *MTA1* possesses the deletion capability on *tf*).

$$\boldsymbol{MTA1} \overset{6}{\mapsto} \boldsymbol{MTA2}:\left\{\begin{array}{c}\mathbf{RMI}(MTA2.receive(\langle \text{content of } tf\rangle,V));\\ \mathbf{Cap}(MTA1;MTA2.receive(*,V));\mathbf{Vouch}(MTA2.receive)\end{array}\right\}$$

The authorization checker located on the host of *MTA2* checks the validity of the capability $\mathbf{Cap}(MTA1;MTA2.receive(*,V))$ and of the voucher $\mathbf{Vouch}(MTA2.receive)$. The voucher contains the capability for *MTA2* to invoke the method *mdeliver* of the object *Vmailbox* corresponding to *V*'s mailbox.

$$\boldsymbol{D_2 \rightarrow JC_2}: \left\{ \begin{array}{c} \mathbf{RMI}\left(MTA2.receive\left(\langle \text{content of } tf \rangle, V\right)\right); \\ \mathbf{Cap}\left(MTA1; MTA2.receive(*,V)\right); \mathbf{Vouch}\left(MTA2.receive\right) \end{array} \right\}$$

$$\mathbf{Vouch}\left(MTA2.receive\right) = \left\{MTA2; VMailbox.mdeliver(*); \mathbf{Cap}\left(MTA2; VMailbox.mdeliver(*)\right)\right\}$$

Finally, *MTA2* calls the *mdeliver* method of *V*'s mailbox (message 7 of Fig.6).

$$\boldsymbol{MTA2} \overset{7}{\mapsto} \boldsymbol{VMailbox}: \left\{\mathbf{RMI}\left(VMailBox.mdeliver\left(\langle \text{content of } tf \rangle\right)\right); \mathbf{Cap}\left(MTA2; VMailbox.mdeliver(*)\right)\right\}$$

D_2 inserts the capability $\mathbf{Cap}\left(MTA2; VMailbox.mdeliver(*)\right)$ in the invocation message. Then D_2 invokes the security kernel located on JC_2 to check that the capability is valid.

6. Conclusion

In this report, we have described the functions of the local authorization checker to be located on every host participating in a MAFTIA application, whether the host is a personal workstation or a server. This authorization checker is implemented by the local JVM, which checks the MAFTIA code signature of all objects loaded in the JVM, by a Java Card, which checks the validity of the capabilities, and by a dispatcher, which is an object located in the JVM, mediating all communications between MAFTIA objects and interfacing the Java Card. The Java Card is considered as sufficiently tamper-resistant to guarantee the integrity of the MAFTIA public key and of the authorization server public key, and the confidentiality and integrity of the key pair dedicated to the user or the server (i.e., the "owner" of the Java Card).

In Section 3, we have stated the security properties required for the different parts of the authorization checker, i.e., the Java Card, the JVM and the dispatcher, as well as the security properties to be granted by the authorization server. Section 4 has presented the authorization checker architecture and its protocols. Finally, Section 5 gave an illustrative example.

The next step of the MAFTIA authorization scheme implementation will be the development of the authorization server.

References

[Abghour & al. 2001]
N. Abghour, Y. Deswarte, V. Nicomette, and D. Powell, *Specification of Authorization Services*, LAAS-CNRS, Toulouse, MAFTIA Project IST 1999-11583 Deliverable D27, LAAS Report 01001, 33 pp., 23 January 2001, available at: <http://www.research.ec.org/maftia/deliverables/D27V13.pdf>.

[Algesheimer & al 2001]
J. Algesheimer, C. Cachin, K. Kursawe, F. Petzold, J. A. Poritz, V. Schoup, and M. Waidner, *MAFTIA: Specification of Dependable Trusted Third Parties*, IBM Research, Zurich Research Laboratory, Zurich (CH), MAFTIA Project IST 1999-11583 Deliverable D26, 98 pp., 22 January 2001, available at: <http://www.research.ec.org/maftia/deliverables/D26-final2.pdf>.

[Cachin & al 2001]
C. Cachin, M. Correia, T. McCutcheon, N. F. Neves, B. Pfitzmann, B. Randell, M. Schunter, W. Simmonds, R. Stroud, P. Veríssimo, M. Waidner, and I. Welch, *Service and Protocol Architecture for the MAFTIA Middleware*, MAFTIA Project IST 1999-11583 Deliverable D23, 92 pp., 25 January 2001, available at: <http://www.research.ec.org/maftia/deliverables/D23final.pdf>.

[Chen 2000]
Z.Chen Java Card Technology for Smart Cards, Addison Wesley, 2000.

[Deswarte & al 2001]
Y. Deswarte, N. Abghour, V. Nicomette, and D. Powell, "An Internet Authorization Scheme using Smartcard-based Security Kernels," *International Conference on Research in Smart Cards (e-Smart 2001)*, Cannes (France), 2001, "*Smart Card Programming and Security*", I. Attali and T. Jensen, Eds., Springer-Verlag, LNCS 2140, pp. 71-82.

[Nicomette & Deswarte 1997]
V. Nicomette and Y. Deswarte, "An Authorization Scheme for Distributed Object Systems," *Proc. Int. Symposium on Security and Privacy*, Oakland, CA, USA, 1997, IEEE Computer Society Press, pp. 21-30.

[Nicomette & Deswarte 1996]
V. Nicomette and Y. Deswarte, "Symbolic Rights and Vouchers for Access Control in Distributed Object Systems," *Proc. 2nd Asian Computing Science Conference (ASIAN'96)*, Singapour, 1996, "*Concurrency and Parallelism, Progamming, Networking, and Security*", J. Jaffar and R. H. C. Yap, Eds., Springer-Verlag, LNCS n°1179, pp. 193-203.