# Towards Remote Policy Enforcement for Runtime Protection of Mobile Code Using Trusted Computing

Xinwen Zhang, Francesco Parisi-Presicce, and Ravi Sandhu

George Mason University, Fairfax, Virginia, USA
{xzhang6, fparisip, sandhu}@gmu.edu

**Abstract.** We present an approach to protect mobile code and agents at runtime using Trusted Computing (TC) technologies. For this purpose, a "mobile policy" is defined by the mobile code originator, and is enforced by the runtime environment in a remote host to control which users can run the mobile code and what kind of results a user can observe, depending on the security properties of the user. The separation of policy specification and implementation mechanism in existing mobile computing platform such as Java Runtime Environment (JRE) enables the implementation of our approach by leveraging current security technologies. The main difference between our approach and existing runtime security models is that the policies enforced in our model are intended to protect the resources of the mobile applications instead of the local system resources. This requires the remote runtime environment to be trusted by the application originator to authenticate the remote user and enforce the policy. Emerging TC technologies such as specified by the Trusted Computing Group (TCG) provide assurance of the runtime environment of a remote host.

## 1 Introduction

Mobile code refers to programs and processes that migrate and execute at remote hosts, so that the execution environments are different for different instances. There is a wide range of mobile applications encompassing autonomous mobile agents which actively travel to remote hosts, Java applets, ActiveX, component software (e.g., COM/DCOM/ COM+ and Servlet/EJB), distributed ad hoc and sensor network applications,etcetera [16].

Runtime environments provide mechanisms to protect the user's and the system's sensitive information by enforcing security policies in a local host. The policies are based on the attributes of the code and of the user who is running it. Possible attributes include code sources, URLs, digital signatures, user groups, roles, and credentials. The two mainstream runtime environments currently adopted in industry are Common Language Runtime (CLR) in .Net and Java Runtime Environment (JRE) in Java. In Java, the security in JDK1.0 and JDK1.1 uses a sandbox model to restrict the access of Java Applets based on code source and digital signature, while in JDK1.2, a user-based access control model is introduced [10,15]. Similar to Java, .Net enforces a code access security model based on code source and location, as well as a role-based security model [16].

H. Yoshiura et al. (Eds.): IWSEC 2006, LNCS 4266, pp. 179–195, 2006.

The protection of mobile applications against malicious hosts and users is a more difficult problem. Current security models in runtime environments are used mainly to enforce the local host's security policy to protect the local system resources. However, there are cases in mobile applications where the originator may have some security requirements to protect the sensitive information brought or accessible by the mobile code. For example, a shopping application may carry a user's sensitive information while running in a remote site. The code originator may require that the code can only run in a specific protected domain, and the user who runs this code must have a specific role in an organization, or some other credentials. In this kind of situation, existing access control models for mobile code are not adequate.

In this paper we propose an approach to enforce the policy of the mobile application originator in remote host runtime environments to control accesses from users, by leveraging emerging client-platform-based Trusted Computing (TC) technologies. We call this kind of policy a "mobile policy" in our model, as compared with the remote host's local policy. A mobile policy is the security requirement provided by the originator to specify what kind of subject in a remote host can run this code, execute particular methods/components, or access some sensitive information included with the mobile application. We use the mechanisms in current runtime environments to enforce a mobile policy.

Since the subject of a mobile policy is a user or program that executes or accesses the mobile code in a remote site, the authentication of the subject is a key point to enforce the policy. Java authentication and authorization Service (JAAS) provides a general layer of user-based authentication and access control mechanism, beyond the sandbox model, which can be applied in our approach. One important advantage of our approach is that we try to reuse the runtime security technologies employed in current systems. A prerequisite for it is the basic assumption that all machines on which the code is intended to run guarantee a minimum of security regarding the correct behavior of the runtime environment. For an enterprise-wide environment, this is viable with on-site configuration of each host by the administrator. For multidomain distributed systems, a trusted runtime environment (TRE) is essential for our model. A TRE can be built on a Trusted Computing Base (TCB) and can be considered an extension of TCB. Emerging Trusted Computing (TC) technologies such as TCG's Trusted Platform Module (TPM) which provide hardware-based root of trust and extended trust to upper levels with verifiable platform characteristics, thus enabling remote policy enforcement in our architecture.

Our approach does not exclude ways other than mobile policies to distribute and enforce security requirements in different hosts within an organization. For example, a network administrator could install in each host, at the operating system level, the policy to be used to determine the specific users who can run a specific application. The use of mobile policies with mobile code has many advantages over this approach: (1) as the deployment and management of mobile code and agents is highly automated, the security management should also be automated and flexible, while administrator-involved configuration for individual platforms is burdensome for an organization; (2) extensibility and scalability of access control policy for a mobile application originator since a mobile policy can be updated/revoked easily with our approach; (3) specification

of fine-grained access control for different users with different security properties in the same remote host, by allowing different users in the remote host to obtain different results from this application (beyond the simple "allowed/not allowed to execute); and (4) simplification of the specification and enforcement of global security policies in an organization.

The remainder of this paper is organized as follows: Section 2 shows some examples which can benefit from our approach but are difficult to implement with current runtime security technologies. Section 3 presents an overview of the security model in JRE. Section 4 proposes our trusted platform architecture to support remote policy enforcement in a distributed environment. Section 5 formulates our policy model specification and enforcement in JRE. Section 6 mentions some related work in the mobile code security area, and the differences between these and our approach. Section 7 summarizes this paper and presents our future work.

## 2 Motivating Examples

*Example 1.* In a mobile application intended to perform E-shopping services, the mobile code is transferred to a remote E-commerce server which collects related information, such as price, location, shipping fee, etc., and then returns it to the customer. The code carries the customer's information, such as credit card number, address, telephone number, etc., and some functions to perform specific work, such as data collecting and transporting, order transaction, etc. If the customer makes the decision to order, the mobile code places the order using the customer's information. In this example, the customer has access control requirements that his personal information can only be read by a clerk in a specific organization without modification, and the functions can only be executed in a particular domain. This objective cannot be achieved with current runtime security models based on code attributes and local host's policies to protect the host's resources. Also, the type safety and data encapsulation features of programming languages such as Java cannot solve this problem. With type safe language, a protected variable or class can be declared as a private element in object-oriented programming, but, with this mechanism, the resulting access control is "black or white" to all users, which is not suitable for fine-grained protection.

*Example 2.* Component-based software has been developed and applied in industry so widely that it has become the mainstream for enterprise computing during the last decade. A component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. Regarded as building blocks, components can be reused in many applications and deployed in different places. Consider a credit card company that has implemented a credit service component. The component, with the customer's information as input, will check the database in the credit card server and return some billing information. As a third party software, this component is deployed at an enterprise's application server and applied to build customized applications. As this component accesses the database, the owner of this component (e.g., the credit card company) has to make sure that only an authenticated and properly authorized application developer,

deployer, or user can instantiate it and call it. With current technology, a component deployer or system administrator has some access control mechanisms to do this to some extent. For example, in Enterprise Java Bean (EJB), the deployment descriptor along with the component in the enterprise's application server controls what kind of roles can access the component and can activate its methods. But this XML-based descriptor is not generated by the component owner and cannot reflect his/her fine-grained access control policies, since the component owner normally is not aware of the security context of local roles in the enterprise. In this case, a mobile policy is a better solution, so that whenever the component is initialized and instantiated in the component container, the access control policy from the component owner can be enforced.

## 3 Java Runtime Security

This section presents an overview of the security mechanism in Java Virtual Machine (JVM) for Java mobile code, which is an example that we use to support runtime enforcement of mobile policy in our framework.

### 3.1 Overview

JVM uses the sandbox model to enforce security policies at runtime. The sandbox model in JDK1.0 and JDK1.1 is based on code attributes such as the code's source, the URL, the signature, etc. While JDK1.0 simply prohibits any Java Applet from accessing any of the local system's resources, JDK1.1 assigns to a Java Applet the same permissions as those of a local program if the host can trust the digital signature associated with this applet (reverts to JDK1.0 otherwise). Starting with JDK1.2, the concept of protection domain based on code attributes is introduced with a complex sandbox model, and the Java Authentication and Authorization Service (JAAS) introduces user-based access control, and allows the local system's access control models and policies to be enforced in the runtime environment. Furthermore, a Java policy is augmented by the security policies of the local operating system, for example, to prevent mobile code executed by a user from accessing a file on the hard disk if the same user cannot read the file at the operating system level.

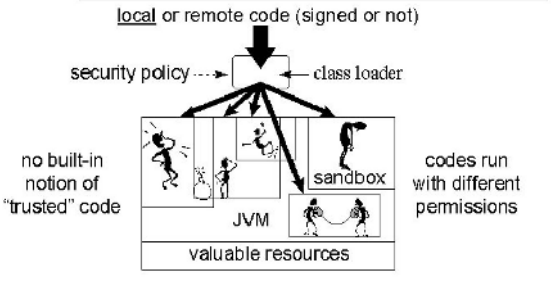


**Fig. 1.** JDK1.2 security model [10]

Figure 1 shows the semantic sandbox model in JDK1.2 [10]. In this model, the code is located in a protection domain which is defined by the code attributes and by the local access control policies. The protection domain represents the permissions that the code can hold during execution. The general process to run Java mobile code can be described schematically as follows: a Java binary code is loaded by a class loader and classes are defined with the *defineClass* method of the class loader. Each class is associated with a protection domain according to policy information. The code is ready to run or be called by other classes after being loaded; whenever it tries to access a local system's resource, it calls a Java API, which then calls the security manager (the access controller since JDK1.2) to check if this operation is allowable. If the security manager permits the operation, the Java API completes the call and returns to the original code, otherwise, the security manger throws an exception to the Java API, which in turn throws it to the user. Starting with JDK1.2, the operation permissions are determined by the access controller, which supplements the security manager.

The access controller in JRE can enforce fine-grained policies based on the attributes of the running code and of the user. Figure 2 illustrates an actual policy file in Java. The permission definition in Java includes two parts: the object and the access right. The objects are the local system's resources, such as the files and directories, sockets, registry keys and values, and so on. Access rights are defined based on object properties, such as "read" to file and directory ("*" means any operation). In Figure 2, the policy allows any code downloaded from "http://www.myuniversity.edu" to "read" files in "/tmp", and to accept connections on, to connect to, or to listen on any port between 1024 and 65535 on any host within "myuniversity.edu". The user has to be authenticated before being defined as "principal" in a policy file and the JAAS provides a mechanism to obtain the authentication context from the local platform. For example, the third item in Figure 2 specifies that "Alice", who is authenticated by Solaris, can access all files and directories within "/usr/home/Alice"; the last item states that an authenticated subject with Kerberos principal name "bob" with realm *foo.org* can call the *System.getProperty* method to access the user environment information. A customized permission class

```
grant codeBase "http://www.myuniversity.edu/"{
   permission java.io.FilePermission "/tmp", "read";
     };
grant signedBy "myuniversity" {
   permission java.net.socketPermission
     "*.myuniversity.edu:1024-",``accept,connect,listen";
   };
grant Principal com.sun.security.auth.SolarisPrincipal
      "Alice" {
      permission java.io.FilePermission
        "/usr/home/Alice", "*";
     }
grant Principal
      javax.security.auth.kerberos.KerberosPrincipal
      "bob@foo.org"  {
      permission java.util.PropertyPermission
        "user.home", "read";
      permission java.io.FilePermission "bar.txt", "read";
};
```

**Fig. 2.** Java policy example

can be defined for an application, thus greatly increasing the flexibility and expressive power of Java security policies.

*Example 3.* Consider the method shown below.

```
Public void sensitiveCall() {
  Permission permission = new
  java.net.SocketPermission("localhost:8080", "connect");
  AccessController.checkPermission(permission);
  // sensitive call
  Socket s = new Socket("localhost", 8080); }
```

In this example, a permission object (*permission*) is defined as a socket connection to local host port 8080. The single-instance class *AccessController* first checks the application's policy file. If this *permission* is granted in the policy or implied by any permission granted in the policy, the *AccessController*'s *checkPermission*[1] method keeps silent; otherwise, an access control exception is thrown to the caller method. Whether a permission is implied by another permission is defined in the *implies* method of the latter's *Permission* or *PermissionCollection* class. The details of defining a customized permission and implied permissions can be found in [9,21]. By default, the *AccessController*'s *checkPermission* method implements the *checkPermission* method implemented in *SecurityManager*.

### 3.2 JAAS

JAAS has been integrated into Java Standard Edition since J2SDK v 1.4. The two purposes of JAAS are to provide user-based authentication and authorization in Java. The original sandbox model in Java is code source-based, so that, a permission is determined by the location where the code comes from and a digital signature generated by the owner. In JAAS, security attributes of the user running the code are considered in access control.

**Authentication.** JAAS implements the Pluggable Authentication Module (PAM) standard with Java. Whenever a mobile application is loaded, the *Configuration* class stores all available *LoginModules* for this application, a *LoginContext* class is instantiated, and its *login* method invokes all *LoginModules* and attempts to authenticate the user. If successful, the user is authenticated as a *Subject* object with a set of *Principals* objects and credentials which represent the user's security attributes. *Principals* are names of identities with particular types, such as a SSN number, a group or domain name, a role, or a tickets. Credentials can be general security related attributes, such as password, public key certificates (X.509 or PGP), Kerberos tickets, etc. For example a successful authentication with *com.sun.security.auth.module.NTLoginModule* imports principals *userID*, *domainID*, and several *groupIDs* for a user.

**Authorization.** Starting with Java 2, the *SecurityManager* delegates security checks to *AccesssController*. After a user is authenticated, the method *Subject.doAs* dynamically associates this user with the *AccessControlContext*, which is retrieved by the *AccessController* to check if it has sufficient permissions for a sensitive operation based on the

[1] Actually this explicit permission check is redundant since any call to open a socket connection is checked by the *SecurityManager* by default.

principals and credentials associated with the subject. A *Subject* class interface has the form

```
Public final class Subject {
    ...
    public static Object doAs (Subject s,
    java.security.PrivilegedAction action) {}
}
```

## 4 Trusted Runtime Environment

As mentioned in Section 1, to correctly enforce a mobile policy, the application originator needs to trust the runtime environment of the remote host. A trusted runtime environment (TRE) should not only detect any malicious modification of the policy, but also detect any change of the security components in the virtual machine, such as authentication and authorization modules. Specifically, a trusted runtime environment (TRE) should provide:

- *Integrity of mobile policy and code.* Before being loaded, a mobile policy's integrity should be attested and verified by the originator (the user who deploys the code) to ensure that the correct policy is used. This requires that the JVM correctly measures the integrity (e.g., with a hash function) and reports to the originator, upon a request to run the code. On the other side, a remote host may also need to verify the integrity and signature of the mobile policy, according to its local policies. For example, the digital signature of a mobile policy/code enables it to be launched in a JVM as a third party policy provider by means of code source-based authorization.
- *Trusted authentication of remote subjects.* The authentication modules in the remote site must authenticate the user in the expected manner. While a uniform approach to authentication may be viable in an organization-wide system, more generally a trust mechanism is needed for multi-domain distributed systems.
- *Trusted authorization enforcement.* After a mobile policy is loaded, the enforcement depends on the expected behavior of the remote JVM's authorization module, which is the policy enforcement point of the security system.

Therefore, a TRE is a prerequisite for our security model. It has been recognized for some time that software alone does not provide an adequate foundation for building a high-assurance trusted platform. The emergence of industry-standard Trusted Computing (TC) technologies promises a revolution in this respect by providing roots of trust upon which secure applications can be developed. These technologies offer a particularly attractive platform for security policy enforcement in general distributed systems. Many current efforts, especially the industry-led Trusted Computing Group (TCG), have focused on building trust rooted in hardware [5].

TCG has defined a set of specifications aimed at providing a hardware-based root of trust and a set of primitive functions that allow trust to propagate to application software, in addition to crossing over platforms. The root of trust in TCG is a hardware component on the platform called the Trusted Platform Module (TPM). Application-level trust requires strong integrity checks of binary code for running processes and a mechanism that allows other entities (applications or platforms) to verify that integrity

as well. A TPM has the capabilities to measure and report runtime configurations of the platform, from BIOS to OS. TPM and TC-enhanced hardware technologies, such as Intel's LaGrande Technology (LT) [2] and AMD's Secure Execution Mode (SEM) [1], generally allocate isolated memory partitions to different application processes to prevent software-based attacks at runtime.

In our work, we abstract the underlying trusted computing technology, and focus on a high-level trusted runtime environment built beyond that. Since a runtime environment such as Java Virtual Machine is normally loaded after the OS is loaded, we consider the TRE as an application or service level trusted domain, which is built beyond the trusted hardware and OS of the remote host with the attestation mechanism of trusted computing technology, as shown in Figure 3. In this platform, the hardware layer (comprising a TCG compliant TPM and some other necessary hardware such as LT-enabled CPU and chipset) provides the root of trust for TC. The secure kernel (SK) provides the protected runtime environment for the JVM. This can be done through controlling DMA-enabled device drivers and memory management unit (MMU).

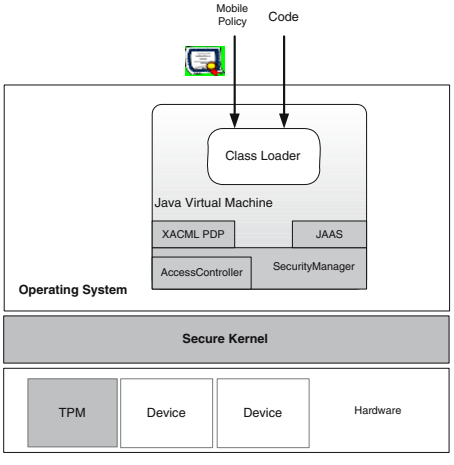


**Fig. 3.** Platform architecture to support trusted runtime environment for mobile code

### 4.1 The Trust Model

The integrity of SK is measured by the TPM when the system boots. Also, SK is protected in memory space by hardware so that its integrity is guaranteed at runtime. Before the JVM is started, SK measures the integrity of JVM and stores its hash value locally. In turn, when mobile code is loaded, the JVM measures the integrity of the program (Java bytecode) and the mobile policy, e.g., implemented by the class loader of JVM. Note that SK enhances the language-based security of the JVM by means of trusted hardware.

The measured integrity can be verified by the code originator with remote attestations, which is enabled by the TC hardware. A hash chain is constructed corresponding

to an attestation challenge to establish the trust of JVM, the mobile code, and the mobile policy based on the root of trust provided by the hardware. Specifically, SK has a public-private key pair generated by the TPM when the platform is initialized, where the public key is certified by the attestation identity key (AIK) of TPM. SK also generates a public-private key pair for the JVM, where the public key is certified by the SK (by signing with its private key) and the private key is protected by JVM, e.g., with the sealed storage function of TPM. The key pair for JVM is generated for the first time when it is installed in the platform. When the platform receives an attestation challenge from a remote side to check a running code's environment state, TPM signs a set of platform configuration register (PCR) values with its AIK key[2], and SK signs the integrity value of JVM with its private key, while JVM signs the integrity value of the code. These three signatures are then sent to the attestation challenger. The challenger verifies all the signatures and the public key certificates of AIK, SK, and JVM, respectively. If all are valid and the integrity values match, the JVM is trusted, and the code and mobile policy's authenticity is verified. Thus, the code originator can trust the security enforcement of the remote JVM and the result generated by the code.

## 5 Mobile Policy Specification and Enforcement

The primary goal of our framework is to enforce the code originator's mobile policy in remote runtime environments. Policy management in our framework includes three phases: (1) policy specification by a mobile code originator, (2) policy distribution by the originator or a trusted third party (such as a central server), and (3) policy enforcement in the remote host. We mainly describe phases (1) and (3) in this paper. For phase (2), a mobile policy could be attached to the code and distributed along the network, in which the policy can be bound to the code itself, or the policy could be downloaded from a central repository only to sites where the code is actually run. In both cases the integrity of the mobile policy is critically important, as mentioned earlier.

### 5.1 Mobile Policy Specification

We have two levels of policy specification. The high-level phase is a logic specification with an authorization specification language (ASL) [14]. This provides a clear definition and analysis, as well as confliction resolution, which is needed when the policies are derived or composed from different resources. For example, a policy can be derived from a policy in a group and another policy of an individual user, or a policy can be combined from several policies from different departments in an organization. The low-level phase is a concrete specification of the mobile policy with the extensible access control markup language (XACML) [3] format, enforced in a runtime environment as an input. The separation of these two levels provides flexible deployment and decentralized policy specification and composition. Because of space limitations we only explain the XACML policy specification in this paper.

[2] We do not explicitly specify what PCR values are included in an attestation, since the required properties of a platform (including hardware, BIOS, and OS configurations) are very application-specific.

XACML is an open-standard format to specify access control policies, and expected to be widely used thanks to the properties of interoperability and extensibility. A mobile policy can be described in XACML format as the following shows:

```
<Policy PolicyId="(policy-name)"
        PolicyCombinationAlg="rule-combining-algorithm:permit-overrides">
  <Target>
    <Subjects>(predicates over subject attributes)</Subjects>
    <Resources>(predicates over object attributes)</Resources>
    <Actions>(predicates over access rights such as read and write)</Actions>
  </Target>
  <Rule effect="permit"/> (Specification that this policy is positive)
  <Obligations>(Specification of attribute-update actions)</Obligations>
</Policy>
```

where the `<Subjects>` and `<Resources>` elements specify the attributes of the subjects and the objects, the rights are in `<Actions>` element, and the update actions are defined in `<Obligations>` element. Update of attributes result from granting the access thereby possibly changing the state of the subject or the object.

**Subjects.** A subject is a process running on behalf of a user or role that actually executes the code. In a mobile policy, subject attributes can be a username, or a role name, group name, certificate signed by a particular certificate authority, etc. Each subject or user attribute has to be authenticated by the runtime platform before running the code. JAAS, entrusted with enforcing the user-based access control, can be used within an enterprise or organization. For general distributed environment, a trusted third party subject attribute service may be needed for authentication.

**Permissions.** A pair (object, right) is regarded as a permission. The objects[3] in a mobile policy may be classes or methods of a mobile code, or information accessed or stored by a mobile code. Specifically, since a mobile policy is to protect a mobile application, possible objects include information on the state of the mobile code, results accumulated at other hosts by a mobile agent, sensitive information of the code originator, and functions to access other sensitive information, implemented as variables, classes, methods or components of a mobile application. Normally, the right associated with a function or component is to "execute", the right for any sensitive information, partial result, and individual variables may include "read" and "write", and the right to a class may be "instantiate" and "inherit". We assume that all the sensitive accesses of (object, right) are encapsulated in a method implemented in the classes, while the sensitive variables are private members of the classes. For example, to "read" a credit card number, a call to *getCredit* method is invoked, while "write" a credit number with *setCredit*. Thus, a permission must be granted to call a method to obtain sensitive information. So generally a permission is checked when a sensitive method is invoked and executed, or a protected object is instantiated or constructed.

### 5.2 Mobile Policy Enforcement

In a typical access control system, a policy decision point (PDP) evaluates access requests with subject and object attributes and sends results to a policy enforcement point

[3] Note that an object in a mobile policy is a different concept from the object (an instance of a class) in Java language.

(PEP). Using Sun's XACML library [4], the PDP module interprets XACML policies in the mobile policy file and makes access decisions, while the PEP can be just a simple interface of the enforcement mechanisms implemented in current JVM (refer to Figure 3). To re-use these functions, each mobile code needs to implement the permission classes for the protected access rights, which are application-specific.

**Define Permission Classes.** Although Java API provides some basic permission classes, most of them are used in local policy enforcement. Normally a mobile code originator has to define his/her own permissions according to the particular applications. For instance, in Example 1, a *creditPermission* class is needed with rights such as "read". Figure 4 is the skeleton to define a *CreditPermission* class for the E-Shopping example. In Java, an application-defined permission class inherits from the system class *Permission* and implements the *Serializable* interface. Each permission object has a type, name and action (access type). For *CreditPermission*, we only define "read" access type. Note that a permission instance does not imply that this permission is granted, but states that accessing this instance is checked by the access controller.

```
public final class CreditPermission extends Permission
 implements Serializable {
 public CreditPermission(string name, string actions){
 //Creates new CreditPermission object with the
 //specified actions. name is the method name that
 //represents the method to read credit card number,
 //such as "getCreditNo". actions is a list of the
 //desired actions granted to the object. In this example,
 //only "read" action to credit information.
    ...
    }
    public boolean implies(Permission permission){...}
    ...
}
```

**Fig. 4.** Sample permission class

The *implies* method specifies complex permission semantics, such as a prerequisite permission. For example, an "update" permission of an online account requires a "read" permission to that account object. Permission constraints such as separation of duty can also be specified in this method.

**Import XACML Mobile Policy.** From the XACML policy file, each subject in the mobile policy is mapped to principals defined in JAAS, such as role, group-name, etc, while the subject attributes and security related credentials such as password, ticket, public key certificate, etc., are associated with these principals after authentication. One of the advantages of using XACML for mobile policy is that XML provides flexible data specifications and semantics, and it is easy to extend it in future work if other information is needed to specify policies. Also, graph tools can be easily developed for policy composition and analysis.

Since the default policy implementation in Java is in a text file, we need to replace this with our alternative implementation. For this, an *XMLPolicy* class is defined which is a subclass of the abstract class *Policy* in Java, and is part of the PDP module to retrieve policy information from the XML file.

A mobile policy is defined by the code originator who is, in general, not the rightholder of the local host system. Therefore the JVM needs the permission from the local host system to load the mobile policy. In implementation, a mobile policy is loaded into a JVM dynamically as the code is loaded. Specifically, a third-party policy implementation can be inserted into a runtime environment by invoking the *setPolicy* method of the *Policy* class. A mobile policy file can be attached with a mobile code in a single Java Archive (JAR) file and captured by JVM, or it can be stored in a central server and a URL argument as the location is provided to load the application code in JVM. If a mobile policy is outside the remote side's domain, dynamically loading the policy requires *runtimePermission* checked by the *AccessController*. This requires that the remote host's default policy be configured to support a third-party policy provider. Code signature for authentication and integrity of the mobile policy is needed according to the host's local policy.

### 5.3 Policy Enforcement

After the permission and policy classes are loaded, and the user is authenticated with JAAS, a sensitive operation can be authorized to a particular subject at runtime. With JAAS, after a user is authenticated with a set of principals, the method *Subject.doAs* dynamically associates all the principals with the local *AccessController* (actually, it is *AccessControlContext* by calling *AccessController.getContext()*). Then, when a sensitive call is requested, the *AccessController* can make a decision based on the pre-defined policy. As shown in Section 3, a *Subject.doAs* method combines an authenticated subject and a *PriviledgedAction* object. Therefore to enforce a mobile policy, all sensitive operations should be encapsulated in *PriviledgedAction* classes. The following example shows a simple implementation.

*Example 4.* Consider an *eshop* mobile application where *creditPermission* is defined by the code originator and policy is specified as the following XACML format.

```
<Policy PolicyId="makeorder-policy"
        PolicyCombinationAlg="rule-combining-algorithm:permit-overrides">
  <Target>
    <Subjects>
      <Subject>
        <!-- The subject identity must include "OU=Org1". -->
        <SubjectMatch MatchId="function:x500Name-match">
          <AttributeValue DataType="string">OU=Org1</AttributeValue>
          <SubjectAttributeDesignator AttributeId="subject-id" DataType="x500Name"/>
        </SubjectMatch>
        <!-- The subject's rolename is PurchaseManager -->
        <SubjectMatch MatchId="function:regexp-string-match">
          <AttributeValue DataType="string">PurchaseManager</AttributeValue>
          <SubjectAttributeDesignator AttributeId="subject-rolename" DataType="string"/>
        </SubjectMatch>
      </Subject>
    </Subjects>
    <Resources>
      <Resource>
        <ResourceMatch MatchId="function:regexp-string-match">
          <AttributeValue DataType="string">creditPermission</AttributeValue>
          <ResourceAttributeDesignator AttributeId="permission-name" DataType="string"/>
        </ResourceMatch>
```

```
      <Resource>
    </Resources>
    <Actions>
      <!-- "GET" represents the read privilege. -->
      <Action>GET</Action>
    </Actions>
  </Target>
  <Rule effect="permit"/>
</Policy>
```

A *subject* is authenticated as a *Org1.PurchaseManager* role and trying to call the sensitive method *getCredit*. The following code shows the outline of the class.

```
public class EShop {
  public static void main(String[] args) {
    ...
    Subject.doAs(aPurchaseManager, new MakeOrder());
    // where aPurchaseManager is an authenticated Subject
    // with a principal of Org1.role named PurchaseManager.
    ...
  }
}
public class MakeOrder implements PrivilegedAction {
    public Object run() {
        ...
        //sensitive call
        String creditCardNo=CreditInfo.getCreditNo();
        ...
    }
}
```

In this example the sensitive code is encapsulated in the *MakeOrder* class, which implements *PriviledgedAction* class. The *CreditInfo* is a static class that stores a credit card information, which can be obtained by some methods. The *getCredit* method is a sensitive operation since as defined in the XML policy file. The *MakeOrder* will trigger an access control check when *getCredit* is called. According to the policy, the permission is granted. The general authorization in a mobile policy is similar to that in enforcing a local policy.

### 5.4 Access Control Algorithm

Java uses a stack-inspection mechanism to enforce the security policy in the runtime environment. In our model, the same stack-inspection mechanism is used, but the access controller checks the permissions based on the mobile policy file. Specifically, for each call in the stack frame, when there is a call to access protected objects in a mobile code, the call is forwarded to the access controller. The access controller determines if the operation is permitted according to the XML mobile policy: if the operation is not permitted, the access controller throws an exception back to the call, which in turn throws it back to the user running the code, otherwise the call completes the operation. Figure 5 shows the access control algorithm. For each call in the stack, the access control algorithm first checks its protection domain. If the target permission is not in the domain, an *AccessControlException* is thrown; otherwise, the algorithm in turn checks if this calling method is declared as a privileged action. If so, and an *AccessControlContext* is provided in the *doPrivileged* method, then the permission is checked with this

*AccessControlContext*, if not, this permission is granted. If a thread is created by a parent thread, the *AccessControlContext* of the parent is associated with the created thread. The permission is checked with the local thread's inherited context if it has not been granted or denied after the first two steps. More details on stack-inspection mechanism can be found in [9,27].

```
Access Control Algorithm:
checkPermission (permission) {
    //loop, from newest to oldest stack frame
     foreach (stackFrame in the stack of current thread) {
        if (stackFrame caller's protection domain does not
            have permission defined in the mobile policy)
        throw AccessControlException;
        else if (stackFrame calling method has been marked
                as privileged action with permission){
            if (an AccessControllerContext context is
                specified in the call to doPrivildged)
                context.checkPermission(permission);
        return; // allow access
        }
        else if(an AccessControlContext inheritedContext
                is inherited when this thread is created)
            inheritedContext.checkPermission(permission);
    return;
    }
}
```

**Fig. 5.** Access control with mobile policy

## 6 Related Work

Security is a basic problem in mobile computing. Generally, there are two distinct areas in mobile code security: (1) protection of the host from malicious mobile code and (2) protection of the mobile code from malicious hosts or users. Researchers have presented several models and mechanisms to deal with malicious code [20,29], such as Sandbox [19,10,15], code signing/code access [16], proof carrying code [17], etc. Protection of mobile code, however, is still an open problem. Vigna [26] proposes an execution tracing technology for mobile agents using cryptographic hash. Yee [28] presents mechanisms to detect tempering by malicious hosts with partial result authentication codes (PARCs) and forward-integrity security policy. Sander and Tschudin [22] formalize a theoretical result aimed at allowing an agent to preserve some secrecy from a malicious host by using encrypted forms of functions in mobile code. Algesheimer et al [6] introduce an approach for securely executing mobile code that relies on a minimally trusted third party. This third party cannot learn anything about the computing with guarantee of privacy and integrity to the code originator. The main difference between our approach and previous work is that we enforce the security policy in the runtime environment of the mobile code. Compatible with existing mechanism, fine-grained access

control policies can be easily implemented in our approach, at the cost of a minimum of trust in the remote runtime environment.

Another line of work is reported in [12], where a Java Secure Execution Framework (*JSEF*) is proposed to support local user specific security policies and a global security policy defined by the administrator. The objective in *JSEF* is still to protect users from erroneous or malicious mobile code, and not to prevent malicious users from improperly accessing or using mobile code. An isolated program execution approach is presented in [18]. The isolation is achieved by delaying a sensitive operation such as file access to a "modification cache" that is invisible to others in the system. While this is practical in isolated applications to protect local system resources, it is not applicable in our approach since we aim at protecting resources brought in by mobile code, which can be not only an object in the virtual machine, but also a remote resource which can be accessed by the mobile code.

Venkatakrishnan et al [25] present a permission "empowering" mechanism to mobile code in the runtime environment instead of restricting the behavior. The scope of this work is still in the range of protecting resources in the local host from mobile code. Cubaleska et al [8] propose a method to build a trusted policy for a mobile agent owner. The policy indicates which host is malicious or not trusted anymore, so that the owner does not deploy mobile agents to these hosts. Since the trusted policy is a posteriori, the solution is useful only for some mobile applications which re-visit previous hosts. In our approach, the mobile policy is enforced in a trusted runtime environment, with no such limitation. Hohl [13] introduces a blackbox model to protect mobile agents from malicious hosts. In this idea, a parallel executable blackbox agent is generated from the original agent, which has a different structure. As declared by the author, this idea only partially solves the malicious host problem. However, our solution can be applied to any mobile code.

A trusted Java Virtual Macine (TrustedVM) is proposed in [11] to capture the behaviors of a remote computing entity. Similar to our approach, the virtual machine itself is attested by signed-hash mechanism. The main difference between this and our approach is that in TrustedVM, policies are used to confine the behavior of the Java program according pre-defined protocols in distributed environments, while the mobile policy in our framework is to protect the execution of mobile code at runtime, that is, the objects in mobile policy are the components of the code itself. Also, our architecture uses hardware-based TC technologies to enhance the security of the language-based JVM in a platform.

## 7 Conclusions

This paper presents a mobile policy framework to protect the information and resources imported by mobile code and agents in runtime environments with trusted computing technologies. This framework includes policy specification and definition, as well as a high-level implementation architecture in Java environment. For the implementation, the access control mechanism in the Java Runtime Environment is used with the existing stack-inspection mechanism. The benefit of this enforcement architecture is that we can define and implement the permission class in a mobile policy, maintaining the

flexibility and compatibility with current runtime technologies. The extensibility of the Java authorization model, as well as the separation of policy specification and enforcement mechanism, makes our approach practical. A trusted computing architecture is proposed in our framework, to provide verifiable trusted behaviors of a remote host's runtime environment.

In future work we can consider development of a runtime policy analysis engine to dynamically answer permission checks. With this, permission derivation and inference, as well as policy analysis can be achieved in runtime. This benefits from scalability and development efficiency beyond the static policy specification and definition. For example, a policy for a code may be combined from several sources, and a real time check and analysis of these sources will improve the system performance by avoiding the redefinition of the static policy files and the restarting of the program.

## References

1. AMD platform for trustworthy computing. Microsoft WinHEC, http://www.microsoft.com/whdc/winhec/pres03.mspx, 2003.
2. *LaGrande Technology Preliminary Architecture Specification, http://www.intel.com/technology/security/downloads/PRELIM-LT-SPEC_D52212.htm.*
3. OASIS XACML TC. *Core Specification: eXtensible Access Control Markup Language (XACML)*, 2005.
4. Sun's XACML implementation, http://sunxacml.sourceforge.net/.
5. *TCG Specification Architecture Overview*. https://www.trustedcomputinggroup.org.
6. J. Algesheimer, C. Cashin, J. Camenisch, and G. Karjoth, Cryptographic Security for Mobile Code, IEEE Symposium On Research in Security and Privacy, 2001.
7. D. Balfanz and L. Gong, Experience with Secure Multi-Processing in Java, International Conference on Distributed Computing Systems, 1998.
8. B. Cubaleska and M. Scheider, Applying Trust Policies for Protecting Mobile Agents Aganist DoS, 3rd Workship on Policies for Distributed Systems and Networks, 2002.
9. L. Gong, E. Gary, and D. Mary, Inside Java 2 Platform Security: Architecture, API Design, and Implementation, Addison-wesley, 2003.
10. L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers, Going Beyond the Sandbox: An Overview of the New Security Arthitecture in the Java Development Kit 1.2, USENIX Symposium on Internet Technologies and Systems, 1997.
11. V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation - a virtual machine directed approach to trusted computing. In *Proc. of the Third virtual Machine Research and Technology Symposium*. USENIX, 2004.
12. M. Hauswirth, C. Kerer and R. Kurmanowytsch, A Secure Execution Framework for Java, In Proc. of ACM Computer and Communication Security, 2000.
13. F. Hohl, Time Limited Blackbox Security: Protecting Mobile Agents From Malicious Hosts, Lecture Notes in Computer Science 1419, Springer-Verlag, Berlin, 1998.
14. S. Jajodia, P. Samarati, and V. Subrahmanian, and E. Bertino, A Unified Framework for Enforcing Multiple Access Control Policies, ACM SIGMOD, 1997.
15. C. Lai, L. Gong, L. Koved, A. Nadalin, and R. Schemers, User Authentication and Authorization in the Java Platform, Annual Computer Security Applications Conference, 1999.
16. B. LaMacchia, S. Lange, M. Lyons, R. Martin, and K. Price, .Net Framework Security, Addison-Wesley, 2002.
17. P. Lee and G. Necula, Research on Proof-carry Code for Mobile Code Security, DARPA workshop on Foundation for Secure Mobile Code, 1997.

18. Z. Liang, V. N. Venkatakrishan, and R. Sekar, Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs, Annual Computer Security Applications Conference, 2003.
19. G. McGraw and E. Felten, Securing Java: Getting Down to Business with Mobile Code, Wiley, http://www.securingjava.com, 1999.
20. G. McGraw and G. Morrisett, Attacking Malicious Code: A Report to the Infosec Research Council, IEEE Software, Volume 17 Issue 5, Sep/Oct 2000.
21. S. Oaks, Java Security, O'Reilly, 2001.
22. T. Sander and C. F. Tschudin, Protecting Mobile Agent against Malicious Hosts, In G.Gigna, ed., Mobile Agents and Security, Lecture Notes in Computer Science 1419, 1998.
23. TCPA Design Philosphies and Concepts, http://www.trustedcomputing.org/home
24. Trusted Computing Group Home, https://www.trustedcomputinggroup.org/home
25. V. Venkatakrishnan, R. Peri, and R. Sekar, Empowering Mobile Code Using Expressive Security Policies, New Security Paradigms Workshop, 2002.
26. G. Vigna, Protecting Mobile Agents Through Tracing, In Proc. of the Workshop on Mobile Object systems, 1997.
27. D. S. Wallach and E. Felten, Understand Java Stack Inspection, IEEE Symposium On Research in Security and Privacy, 1998.
28. B. Yee, A Sanctuary for Mobile Agents, Secure Internet Programming: Security Issures for Mobile Code and Distributed Objects, J.Vitek and C.Jensen, eds., LNCS 1603, 1999.
29. J. Zachry, Protecting Mobile Code in the Wild, IEEE Internet Computing, March/April 2003.