Preserving Privacy in Distributed Delegation with Fast Certificates

Pekka Nikander[†], Yki Kortesniemi[‡], Jonna Partanen[‡]

[†] Ericsson Research
FIN-02420 Jorvas, Kirkkonummi, Finland
pekka.nikander@ericsson.com

[‡] Helsinki University of Technology, Department of Computer Science,[1]
FIN-02015 TKK, Espoo, Finland
yki.kortesniemi@hut.fi, jonna.partanen@hut.fi

Abstract. In a distributed system, dynamically dividing execution between nodes is essential for service robustness. However, when all of the nodes cannot be equally trusted, and when some users are more honest than others, controlling where code may be executed and by whom resources may be consumed is a nontrivial problem. In this paper we describe a generic authorisation certificate architecture that allows dynamic control of resource consumption and code execution in an untrusted distributed network. That is, the architecture allows the users to specify which network nodes are trusted to execute code on their behalf and the servers to verify the users' authority to consume resources, while still allowing the execution to span dynamically from node to node, creating delegations on the fly as needed. The architecture scales well, fully supports mobile code and execution migration, and allows users to remain anonymous.

We are implementing a prototype of the architecture using SPKI certificates and ECDSA signatures in Java 1.2. In the prototype, agents are represented as Java JAR packages.

1 Introduction

There are several proposals for distributed systems security architectures, including the Kerberos [14], the CORBA security architecture [23], and the ICE-TEL project proposal [6], to mention but a few. These, as well as others, differ greatly in the extent they support scalability, agent mobility, and agent anonymity, among other things. Most of these differences are clearly visible in the trust models of the systems, when analyzed.

In this paper we describe a Simple Public Key Infrastructure (SPKI) [7] [8] [9] based distributed systems security architecture that is scalable and supports agent mobility, migration and anonymity. Furthermore, all trust relationships in our architecture are explicitly visible and can be easily analyzed. The architecture allows various security policies to be explicitly specified, and in this way, e.g., to specify where an agent may securely execute [27].

Our main idea is to use dynamically created SPKI authorisation certificates to delegate permissions from an agent running on one host to another agent running on another host. With SPKI certificates, we are able to delegate only the minimum rights the receiving agent needs to perform the operations that the sending agent wants it to carry out. The architecture allows permissions to be further delegated as long as the generic trust relationships, also presented in the form of SPKI certificates, are preserved.

A typical application could be a mobile host, such as a PDA. Characteristic to such devises are limited computational power, memory constraints and an intermittent, low bandwidth access to the network. These pose some limitations on the cryptographic system used. Favourable characteristics would be short key length and fast operation with limited processing power.

In order to be able to distinguish running agents, and delegate rights to them, new cryptographic key pairs need to be created, and new certificates need to be created and verified. To

[1] This work was partially funded by the TeSSA research project at Helsinki University of Technology under a grant from TEKES.

make this happen with an acceptable speed, we have implemented the relevant public key functions with Elliptic Curve based DSA (ECDSA), yielding reasonable performance.

In our architecture, cryptographic key pairs are created dynamically to represent running agents. This also has a desirable side effect of making anonymous operations possible while still preserving strong authorisation. In practical terms, some of the certificates that are used to verify agent authority may be encrypted to protect privacy. This hinders third parties, and even the verifying host, from determining the identity of the principal that is responsible for originally initiating an operation. This allows users' actions to remain in relative privacy, while still allowing strong assurance on whether an attempted operation is authorised or not.

We are in the process of implementing a practical prototype of our architecture. The prototype is based on distributed Java Virtual Machines (JVM) running JDK 1.2, but the same principles could be applied to any distributed system. The main parts of the prototype architecture are already implemented, as described in [15], [21], and [25], while others are under way.

The rest of this paper is organized as follows. In Sect. 2 we describe the idea of authorisation certificates, their relation to trust relationships and certificate loops, and the security relevant components of the SPKI certificates. Sect. 3 summarizes the dynamic nature of the SPKI enhanced JDK 1.2 security architecture. Next, in Sect. 4, we describe how our ECDSA implementation complements the Java cryptography architecture. In Sect. 5, we define the main ideas of our architecture, and show how SKPI certificates and dynamically generated key pairs can be used to anonymously, but securely, delegate permissions from one JVM to another. Sect. 6 describes the current implementation status, and Sect. 7 includes our conclusions from this research.

2 Authorisation and Delegation

The basic idea of authorisation, as opposed to simple (identity) authentication, is to attest that a party, or an agent, is authorised to perform a certain action, rather than merely confirm that the party has a claimed identity. If we consider a simple real life example, the driver's licence, this distinction becomes evident. The primary function of a driver's licence is to certify that its holder is entitled, or authorised, to operate vehicles belonging to certain classes. In this sense, it is a device of authorisation. However, this aspect is often overseen, as it seems obvious, even self-evident, for most people.

The secondary function of a driver's licence, the possibility of using it as an evidence of identity, is more apparent. Yet, when a police officer checks a driver's licence, the identity checking is *only* a necessary side step in assuring that the operator of a vehicle is on legal business.

The same distinction can and should be applied to computer systems. Instead of using X.509 type identity certificates for authenticating a principal's identity, one should use authorisation certificates, or signed credentials, to gain assurance about a principal's permission to execute actions. In addition to a direct authorisation, as depicted in the driver's licence example, in a distributed computer system it is often necessary to delegate authority from a party to a next one. The length of such delegation chains can be pretty long on occasions. [17]

2.1 Trust and Security Policy

Trust can be defined as a belief that an agent or a person behaves in a certain way. Trust to a machinery is usually a belief that it works as specified. Trust to a person means that even if that person has the possibility to harm us, we believe that he or she chooses not to. The trust requirements of a system form the system's trust model. For example, we may need to have some kind of trust to the implementor of a software whose source code is not public, or trust to the person with whom we communicate over a network.

Closely related to the concept of trust is the concept of policy. A security policy is a manifestation of laws, rules and practices that regulate how sensitive information and other resources are managed, protected and distributed. Its purpose is to ensure that the handled information remains confidential, integral and available, as specified by the policy. Every agent may be seen to function under its own policy rules.

In many cases today, the policy rules are very informal, often left unwritten. However, security policies can be meaningful not only as internal regulations and rules, but as a published document which defines some security-related practices. This could be important information when some outsider is trying to decide whether an organization can be trusted in some respect. In this kind of situation it is useful to define the policy in a systematic manner, i.e., to have a formal policy model.

Another and a more important reason for having a formally specified policy is that most, or maybe even all, of the policy information should be directly accessible by the computer systems. Having a policy control enforced in software (or firmware) rather than relying on the users to follow some memorized rules is essential if the policy is to be followed. A lot of policy rules are already implicitly present in the operating systems, protocols, and applications, and explicitly in their configuration files. Our mission includes the desire to make this policy information more explicit, and make it possible to manage it in a distributed way.

2.2 Certificates, Certificate Chains, and Certificate Loops

A certificate is a signed statement about the properties of some entity. A certificate has an issuer and a subject. Typically, the issuer has attested, by signing the certificate, its belief that the information stated in the certificate is true. If a certificate states something about the issuer him or herself, it is called a self-signed certificate or an auto-certificate, in distinction from other certificates whose subject is not the issuer.

Certificates are usually divied in two categories: Identity certificates and authorisation certificates. An identity certificate usually binds a cryptographic key to a name. An authorisation certificate, on the other hand, can make a more specific statement; for example, it can state that the subject entity is authorised to have access to a specified service. Furthermore, an authorisation certificate does not necessarily need to carry any explicit, human understandable information about the identity of the subject. That is, the subject does not need to have a name. The subject can prove its title to the certificate by proving that it possesses the private key corresponding to the certified public key; indeed, that is the only way a subject can be trusted to be the (a) legitimate owner of the certificate.

Certificates and trust relationships are very closely connected. The meaning of a certificate is to make a reliable statement concerning some trust relationship. Certificates form chains, where a subject of a certificate is the issuer of the next one. In a chain the trust propagates transitively from an entity to another. These chains can be closed into loops, as described in [17].

The idea of certificate loops is a central one in analyzing trust. The source of trust is almost always the checking party itself. A chain of certificates, typically starting at the verifying party and ending at the party claiming authority, forms an open arc. This arc is closed into loop by the online authentication protocol where the claimant proves possession of its private key to the verifying party.

2.3 Authorisation and Anonymity

In an access control context, an authorisation certificate chain binds a key to an operation, effectively stating that the holder of the key is authorised to perform the operation. A run time challenge operates between the owner of operation (the reference monitor) and the key, thus closing the certification loop. These two bindings, i.e., the certificate chain and the run time authentication protocol, are based on cryptography and can be made strong.

In an authorisation certificate, a person-key binding is different from the person-name binding used in the identity certificates. By definition, the keyholder of a key has sole possession of the private key. Therefore, the corresponding public key can be used as an identifier (a name) of the keyholder. For any public key cryptosystem to work, it is essential that a principal will keep its private key to itself. So, the person is the only one having access to the private key and the key has enough entropy so that nobody else has the same key. Thus, the identifying key is bound tightly to the person that controls it and all bindings are strong. The same cannot be claimed about human understandable names, which are relative and ambiguous [10].

However, having a strong binding between a key and a person does not directly help the provider of a controlled service much. The provider does not know if it can trust the holder of the key. Such a trust can only be acquired through a valid certificate chain that starts at the provider

itself. The whole idea of our architecture centres around the concept of creating such certificate chains when needed, dynamically providing agents the permissions they need.

The feature of not having to bind keys to names is especially convenient in systems that include anonymity as a security requirement. It is easy for a user to create new keys for such applications, while creating an authorised false identity is (hopefully) not possible.

2.4 SPKI Certificates

The Simple Public Key Infrastructure (SPKI) is an authorisation certificate infrastructure being standardized by the IETF. The intention is that it will support a range of trust models. [7] [8] [9]

In the SPKI world, principals are keys. Delegations are made to a key, not to a keyholder or a global name. Thus, an SPKI certificate is closer to a "capability" as defined by [16] than to an identity certificate. There is the difference that in a traditional capability system the capability itself is a secret ticket, the possession of which grants some authority. An SPKI certificate identifies the specific key to which it grants authority. Therefore the mere ability to read (or copy) the certificate grants no authority. The certificate itself does not need to be as tightly controlled.

In SPKI terms, a certificate is basically a signed five tuple (**I**,**S**,**D**,**A**,**V**) where

- **I** is the Issuer's (signers) public key, or a secure hash of the public key,
- **S** is the Subject of the certificate, typically a public key, a secure hash of a public key, a SDSI name, or a secure hash of some other object such as a Java class,
- **D** is a Delegation bit,
- **A** is the Authorisation field, describing the permissions or other information that the certificate's Issuer grants to or attests of the Subject,
- **V** is a Validation field, describing the conditions (such as a time range) under which the certificate can be considered valid.

The meaning of an SPKI certificate can be stated as follows:

Based on the assumption that **I** has the control over the rights or other information described in **A**, **I** grants **S** the rights/property **A** whenever **V** is valid. Furthermore, if **D** is true and **S** is a public key (or hash of a public key), **S** may further delegate the rights **A** or any subset of them.

2.5 Access control revisited

The traditional way of implementing access control in a distributed system has been based on authentication and Access Control Lists (ACLs). In such a system, when execution is transferred from one node to another, the originating node authenticates itself to the responding node. Based on the identity information transferred during the authentication protocol, the responding node attaches a local identifier, i.e., an user account, to the secured connection or passed execution request (e.g., an RPC call). The actual access control is performed locally by determining the user's rights based on the local identifier and local ACLs.

In an authorisation based system everything works differently. Instead of basing access control decisions on locally stored identity or ACL information, decisions are based on explicit access control information, carried from node to node. The access rights are represented as authorisation delegations, e.g., in the authorisation field of an SPKI certificate. Because the certificates form certificate loops, the interpreter of this access control information is always the same party that has initially issued it. The rights may, though, have been restricted along the delegation path.

In Sect. 5 we show how this kind of an infrastructure can be effectively extended to an environment of mobile agents, represented as downloadable code, that is run on a network of trusted and untrusted execution nodes.

3 An SPKI based Dynamic Security Architecture for JDK 1.2

As described in more detail in [25], we have extended the JDK 1.2 security architecture with SPKI certificates. This makes it possible to dynamically modify the current security policy rules applied at a specific Java Virtual Machine (JVM). This dynamic modification allows an agent

running on one trusted JVM to delegate permissions to another agent running on another trusted JVM.

The components of the basic and SPKI extended access control architecture are enumerated in Table 1 and discussed in more detail in Sections 3.1-3.2. The most relevant changes needed to the basic architecture are described in Sect. 3.2.

Table 1: The parts of the JDK 1.2 access control Architecture

Class or classes	The role of the class or classes
`Permission` and its subclasses	Represent different “tickets” or access rights.
ProtectionDomain	Connects the Permission objects to classes.
`Policy` and its subclasses	Decide what permissions each class gets.
AccessController	The reference monitor. [1]

3.1 Access Control in JDK 1.2

The JDK 1.2 has a new, capability based access control architecture. Java capabilities are objects called permissions. Each protected resource in the system has a corresponding permission object that represents access to the resource. There are typically many instances of a given permission, possessed by and thus granting access for different classes.

Permissions are divided into several subtypes that extend the Permission class. Each resource type or category, such as files or network connections, has its own Permission subclass. Inside the category, different instances of the Permission class correspond to different instances of the resource. In addition, the programmers may provide their own Permission subclasses if they create protected resources of their own.

Just as in any capability-based access control system, the Java classes must be prevented from creating permissions for themselves and thus gaining unauthorised access. This is done by assigning the classes to protection domains. Each class belongs to one and only one protection domain. Each ProtectionDomain object has a PermissionCollection object that holds the permissions of that domain. Only these permissions can be used to gain access to resources. The classes cannot change their protection domain nor the PermissionCollection of the domain. Thus, the classes are free to create any Permission objects they like, but they cannot affect the access control decisions and gain unauthorised access.

The actual access control is done by an object called AccessController. When a thread of execution requests access to a protected resource such as a file, the AccessController object is asked whether the access is granted or not. To determine this, the AccessController checks the execution context to see if the caller and all the previous classes in the call chain have the Permission object corresponding to the resource. The previous classes in the call chain are checked to ensure that a class does not bypass the access control simply by calling another class with more permissions.

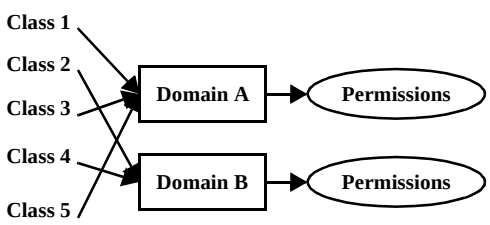

Fig. 1. Classes, domains and permissions

3.2 Policy Management

A security policy defines the rules that mandate which actions the agents in the system are allowed or disallowed to do [1]. Java security policy defines what permissions each protection domain gets. The objects implementing the security policy management in JDK are subclasses of the Policy class. The implementation can be changed easily by just creating and installing a new Policy subclass.

The default policy implementation of JDK 1.2 uses a set of configuration files to define the security policy. This system has several small defects discussed in [21] and [25]. Furthermore, this approach makes delegating permissions from a class in one JVM to another class in some other JVM virtually impossible, as the delegating party should be able to edit the configuration file of the other JVM. We have solved these problems by replacing the configuration files with a capability-based policy definition that uses SPKI certificates to represent capabilities.

In our model, the policy manager and the dynamic permission evaluation are slightly more complex than in the basic implementation. In the SPKI extended system, the main task of the policy manager is to attempt to reduce a set of SPKI certificates to form a valid chain from its own key, called the Self key, to the hash of the classes composing a protection domain, and to interpret the authorisation given by the chain into Java Permission objects. This chain reduction includes checking the validity of the certificates, checking that all but the last certificate have the delegation bit set, and intersecting the authorisation fields to get the final authorisation given by the chain.

In the default JDK implementation, the ProtectionDomains get the permissions when they are initialized, and the permissions are not revised after that. We have made the policy evaluation more dynamic. When a class tries to access a protected resource, the reference monitor asks the protection domain whether it contains the specific permission required, and the protection domain in turn asks the Policy for the permission. The Policy will try to produce a certificate chain reduction that would imply the permission in question. If it fails, the access is not granted.

The SPKI drafts propose that the Prover (i.e. the class) is responsible of presenting a valid certificate chain to the Verifier (i.e. the Policy) at the time of access request or authentication [7]. We argue that this approach does not work with mobile agents. Requiring that each mobile agent includes the logic for locating all certificates needed to access resources is infeasible and counterproductive. Instead, we think that the Policy will need to locate the relevant certificates as well as to reduce the certificate chains.

4 Adding Elliptic Curve based Certificates to Java

Java defines and partially implements security related functionality as part of its core API. This functionality is collected in the `java.security` package and its subpackages. To facilitate and co-ordinate the use of cryptographic services, JDK 1.1 introduced the Java Cryptography Architecture (JCA). It is a framework for both accessing and developing new cryptographic functionality for the Java platform. JDK 1.1 itself included the necessary APIs for digital signatures and message digests.[7]

In Java 1.2, JCA has been significantly extended. It now encompasses the cryptography related parts of the Java Security API, as well as a set of conventions and specifications. Further, the basic API has been complemented with the Java Cryptography Extension (JCE), which includes further implementations of encryption and key exchange functionality. This extension, however, is subject to the US export restrictions and is therefore not available to the rest of the world. To fully utilise Java as a platform for secure applications, the necessary cryptographic functionality has to be developed outside the US.

4.1 The Java Cryptography Architecture

One of the key concepts of the JCA is the provider architecture. The key idea is that all different implementations of a particular cryptographic service conform to a common interface. This makes these implementations interchangeable; the user of any cryptographic service can choose whichever implementation is available and be assured that his application will still function.

To achieve true interoperability, Java 1.2 defines cryptographic services in an abstract fashion as engine classes. The following engine classes, among others, have been defined in Java 1.2:

- MessageDigest – used to calculate the message digest (hash) of given data
- Signature – used to sign data and verify digital signatures
- KeyPairGenerator – used to generate a pair of public and private keys suitable for a specific algorithm
- CertificateFactory – used to create public key certificates and Certificate Revocation Lists (CRLs)
- AlgorithmParameterGenerator – used to generate a set of parameters to be used with a certain algorithm

A generator is used to create objects with brand-new contents, whereas a factory creates objects from existing material.

To implement the functionality of an engine class, the developer has to create classes that inherit the corresponding abstract Service Provider Interface (SPI) class and implement the methods defined in it. This implementation then has to be installed in the Java Runtime Environment (JRE), after which it is available for use.[7] [8]

4.2 Implementing an Elliptic Curve Cryptography Provider in Java 1.2

In our project we implemented the Elliptic Curve Digital Signature Algorithm (ECDSA). The signature algorithm and all the necessary operations are defined in IEEE P1363 and ANSI X9.62 drafts. To facilitate the interoperability of different implementations, Java 1.2 includes standard names for several algorithms in each engine class together with their definitions. ECDSA, however, is not among them. We therefore propose that ECDSA should be adopted in Java 1.2 as a standard algorithm for signatures.

Similarrly with the DSA implementation in JDK 1.2, we have defined interfaces for the keys, algorithm parameters (curves) and points. These are used to facilitate the use of different co-ordinate representations and arithmetics. Our implementation of ECDSA uses prime fields and affine co-ordinates. The mathematics have been implemented using the BigInteger-class. The BigInteger class is easy to use and flexible as it implements several operations necessary for modular arithmetic and provides arbitrary precision. The down side is that performance is not optimal. If the key length could be kept small enough, the arithmetic could be based on the `long` type. The necessary operations could be based on using a few `long` type variables for each value. With regular elliptic curves, which require a key length of at least 160 bits, this approach might be inconvenient, but if hyperelliptic curves were used, the approach could prove feasible.

Even further improvements in performance could be achieved by implementing the key mathematic operation in the hardware, e.g., in a mobile host. With the small key size of (hyper)elliptic curves, this would not pose unreasonable demands on the processor design or memory.

5 Extending Java Protection Domains into Distributed Agents

The dynamic and distributed nature of the SPKI based Java protection domains, described in Sect. 3, opens up new possibilities for their use. In particular, it is possible to dynamically delegate a permission from one domain, executing on one Java virtual machine, to another domain, executing on another Java virtual machine. For example, when a distributed application requests a service from a server, it might want to allow a certain class, an agent, in the server to execute as if it were the user that started the application in the first hand. This ability allows us to view the protection domains not just as internal Java properties, but they can be considered to represent active agents that are created and executed in the network.

In order to be able to perform these kinds of functions, the domains (or agents) involved must have local access to some private keys, and a number of trust conditions must be met. The requirement of having access to a private key can be easily accomplished by creating a temporary key pair for each policy domain, i.e., for each incarnation of an agent. This is acceptable from a security point of view, because the underlying JVM must be trusted anyway, and so it can be trusted to provide temporary keys as well. The public temporary key can be signed by the local machine key, denoting it as belonging to the domain involved.

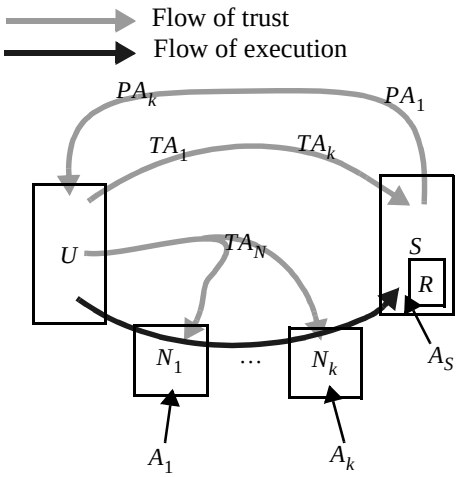

Fig. 2. The user U requests for a service needing the resource R through intermediate nodes N_1, …, N_k.

To analyze the trust conditions, let us consider the situation depicted in Fig. 2. The user U wants to use a protected resource R, located on the server S. However, we assume that it is not possible or feasible that the user U would have a direct secured connection with S. As an example application, the user may be using a mobile terminal whose connectivity cannot be guaranteed. So, instead of a direct connection the user's actions are carried out by one or more intermediate nodes N_i, each acting on the user's behalf.

The setting is still slightly more complicated by the assumption that the code that actually executes at the server S and the intermediate nodes N_i consist of independent agents, which are dynamically loaded as needed. In practical terms, in our prototype these agents are Java class packages (jar files), carrying SPKI certificates within themselves. The agents are named as A_S for the agent eventually running at the server S, and as A_i for the agents running at the intermediate nodes N_i.

It is crucial to note that when the execution begins, the user U typically does not know the identity of the server S, the intermediate nodes N_i, or the agents A_S, A_i. Instead, she has expressed her confidence towards a number of administrators (described below), who in turn certify the trustworthiness of S and N_i. Correspondingly, the server S has no idea about the user U or the nodes N_i. Again, it trusts a number of administrators to specify an explicit security policy on its behalf.

5.1 Trust requirements

Since we assume that the nodes in the network do not necessarily nor implicitly trust each other or the executable agents, a number of trust conditions must be met and explicitly expressed.

First, from the user's point of view, the following conditions must be met.

- The user U must trust the server S to provide the desired service S_R granting access to the resource R. This trust is expressed through a sequence of trust administrators TA_i, where the last administrator TA_k confirms that S indeed is a server that provides the service S_R.
- The user U must trust the agent A_S, and delegate the right of accessing the resource R to it. However, the actual runtime identity (i.e, the temporary public key) of the particular activation of A_S, running on S on the behalf of U on this occasion, is not initially known but created runtime. On the other hand, U must certify the code of A_S so that it may be loaded on her behalf.
- The user U must consider each of the intermediate nodes N_i to be trustworthy enough to execute code on and to participate in accessing the resource R on her behalf. For simplicity, in this case we have assumed that the trustworthiness of the nodes is certified by a single trust authority TA_N, directly trusted by the user U.
- The user U must trust the intermediate agents A_i, while running on the nodes N_i, to execute on her behalf and to participate in the process. Again, the temporary public keys of the actual incarnations of the agents are created only at runtime.

From the server's point of view, a number of similar conditions must be met.

- The user U must be authorised to access the resource R. Since the resource R is controlled by the server S, the source of this authority must be S itself. Typically, this authorisation is achieved through a chain of independent security policy administrators PA_i.
- The server S must trust the intermediate nodes N_i to faithfully represent the user U[1]. This means, among other things, that when an agent is running on any of these nodes, S trusts that the node has faithfully created and certified the temporary key pair that represents the agent. For simplicity, we have assumed that the server S assumes the user U to be competent enough to determine which nodes to trust. Thus, in practice, the certificate chain used to delegate the right to access the resource R may be combined with the chain certifying U's proficiency in determining node trustworthiness.

5.2 Expressing the Trust Requirements with SPKI Certificates

Using SPKI certificates, it is possible to explicitly express the static and dynamic trust and delegation relationships. In the following, the appearance of the symbols U, S, N_i, TA_i, TA_N and PA_i as the issuer or the subject of the certificates denotes the (static) public key of the respective principal. On the other hand, to explicitly communicate the dual nature of the agents as dynamically loaded code and dynamically created key pairs that represent them, $h(A)$ denotes a hash code calculated over the code of the agent A, and $K_{A,N}$ denotes a temporary key that the node N has created for the agent A. Furthermore, the symbol R is used to denote the permission to access the resource R.

Normal SPKI certificates are represented as 4-tuples (I, S, D, A), where the validity field is left out. Correspondingly, SPKI name certificates are represented as $((I's\ name), S)$, denoting that the issuer I has bound the *name* for the principal S.

User trust requirements. First, U's trust on S is represented through a certificate chain Cert. 1 ... Cert. 3.

$$(U, TA_1, true, S_R) \qquad \text{Cert. 1}$$

$$\ldots \qquad \text{Cert. 2}$$

$$(TA_k, S, false, S_R) \qquad \text{Cert. 3}$$

Second, U must further certify that the agents, when run, may use whatever rights U has granted to the agents as code. Since U does not know where the agents will be run, SPKI certificates containing indirect naming are used to denote this delegation.

$$(U, (U's\ N's\ h(A_i)), false, \text{act as } h(A_i)) \qquad \text{Cert. 4}$$

where $(N's\ h(A_i))$ is an SPKI name denoting the running agent A_i, running on an arbitrary node N, named by U.

Next, U must certify that the nodes are trustworthy to execute code. U has delegated this right to TA_N; thus, a chain of two certificates is needed for each node. In practice, the right of running code on the issuer's behalf is represented by a number of SPKI naming certificates that transfer the node name N, used above, from U's name space to the name space of the trust authority TA_N. The trust authority TA_N, on it's behalf, names a specific node N_i as a node N, which, consecutively, has the authority to bind the agent hash $h(A_i)$ to a public key.

$$((U's\ N), (TA_N's\ N)) \qquad \text{Cert. 5}$$

$$((TA_N's\ N), N_i) \qquad \text{Cert. 6}$$

Furthermore, the user U must certify the actual code of the agents A_i. In a real situation, this would happen through another certificate chain. However, for simplicity, we assume that the user has written the agents herself, and therefore certifies their code directly.

$$(U, h(A_i), true, R) \qquad \text{Cert. 7}$$

[1] More generally, the server S must trust the intermediate nodes to faithfully represent any user, or at least any user that has the authority and a need to access the resource R.

Server trust requirements. Similar to the user, the server S must authorise the user U to access the resource R, represented as the chain Cert. 8 ... Cert. 10.

$$(S, PA_1, true, R) \quad \text{Cert. 8}$$

$$\ldots \quad \text{Cert. 9}$$

$$(PA_k, U, true, R) \quad \text{Cert. 10}$$

Since the user is allowed to directly denote which nodes she trusts, no other certificates are needed on the server's behalf.

Initial reductions. Reducing Certificates 1–3, one gets the certificate

$$(U, S, false, S_R) \quad \text{Cert. 11}$$

This is sufficient for the user, and to anybody acting on the user's behalf, to verify that the server S really provides the desired service S_R, which allows one to access the resource R.

Respectively, reducing the Certificates 4–6, the result is

$$(U, (N_i's\ h(A_i)), false, \text{act as } h(A_i)) \quad \text{Cert. 12}$$

denoting that the user U has delegated to the agent A_i, as named by the node N_i, the right to use the rights assigned to the agent's code[1].

5.3 Runtime Behaviour

The run time permission delegation is advanced step by step, from the user through the intermediate nodes to the server. We next describe the initial step, a generic intermediate step, and the final step at the server.

Initiation of action. As the user U initiates her access, she contacts the first intermediate node N_1. The node loads the agent A_1, generates a temporary key K_{A_1, N_1} for the agent, and creates an SPKI name certificate (Cert. 13) to name the agent.

$$((N_1's\ h(A_1)), K_{A_1, N_1}) \quad \text{Cert. 13}$$

Reducing this with Cert. 12 gives the newly created key the acting right.

$$(U, K_{A_1, N_1}, false, \text{act as } h(A_i)) \quad \text{Cert. 14}$$

Combining this, on the semantic level[2], with Certificates 7–10, results in the creation of Cert. 15 that finally denotes that the newly created key has the S delegated permission to access R, and to further delegate this permission.

$$(S, K_{A_1, N_1}, true, R) \quad \text{Cert. 15}$$

Intermediate delegation. Let us next consider the situation where the node N_i has gained the access right.

$$(S, K_{A_i, N_i}, true, R) \quad \text{Cert. 16}$$

The node initiates action on the next node, N_{i+1}, that launches and names the agent running on it.

$$((N_{i+1}'s\ h(A_{i+1})), K_{A_{i+1}, N_{i+1}}) \quad \text{Cert. 17}$$

Reducing this with the chain leading to Cert. 12 results in

$$(U, K_{A_{i+1}, N_{i+1}}, false, \text{act as } h(A_{i+1})) \quad \text{Cert. 18}$$

Having this, together with the Cert. 12 chain, A_i can be sure that it is fine to delegate the right expressed with Cert. 16 further to A_{i+1}.

$$(K_{A_i, N_i}, K_{A_{i+1}, N_{i+1}}, true, R) \quad \text{Cert. 19}$$

Combining Cert. 19 with Cert. 16 results in

[1] The reader should notice that this, naturally, allows N_i to delegate this right to itself. However, this is acceptable and inevitable, as the node N_i is trusted for creating and signing the agent's public key.

[2] With semantic level we mean here that mere syntactic SPKI reduction is not enough, but that the interpreter of the certificates must interpret the expression "act as $h(A_i)$".

$$(S, K_{A_{i+1}, N_{i+1}}, true, R) \quad \text{Cert. 20}$$

which effectively states that A_{i+1}, running on node N_{i+1}, is permitted to access the resource R and to further delegate this permission.

Final step. In the beginning of the final step, agent A_k, executing on node N_k, has gained the right to access R.

$$(S, K_{A_k, N_k}, true, R) \quad \text{Cert. 21}$$

Agent A_k now launches agent A_S to run on the server S. S creates a temporary key K_{A_S} for the agent, and publishes it as a certificate.

$$((S's\ h(A_S)), K_{A_S}) \quad \text{Cert. 22}$$

Again, combining this with the Cert. 12 chain gives

$$(U, K_{A_S}, false, \text{act as } h(A_S)) \quad \text{Cert. 23}$$

which allows the agent A_k to decide to delegate the right to access the resource R.

$$(K_{A_k, N_k}, K_{A_S}, false, R) \quad \text{Cert. 24}$$

Reducing Cert. 24 with Cert. 21 results in Cert. 25.

$$(S, K_{A_S}, true, R) \quad \text{Cert. 25}$$

The final certificate, Cert. 25, can now be trivially closed into a certificate loop by S, since S itself has created the key K_{A_S}, and therefore can trivially authenticate it. In other words, this can be seen easily to reduce into a virtual self-certificate Cert. 26.

$$(S, S, false, R) \quad \text{Cert. 26}$$

Cert. 26, closed on the behalf of the agent A_S, finally assures the server S that the agent A_S does have the right to access the protected resource R.

5.4 Preserving privacy

Using SPKI Certificate Reduction Certificates (CRC) provides the user U a simple way to stay anonymous while still securely accessing the resource R. If any of the policy administrators PA_i on the trust path leading from S to U is available online and willing to create CRCs, the user can feed it the relevant items of Cert. 9, Cert. 10, and Certs 4–6 and Cert. 7. This allows the policy administrator PA_i to create CRCs Cert. 27 and Cert. 28, for Certs 4–6 and Cert. 7, respectively.

$$(PA_i, (N_i's\ h(A_i)), false, \text{act as } h(A_i)) \quad \text{Cert. 27}$$

$$(PA_i, h(A_i), true, R) \quad \text{Cert. 28}$$

Then, in the rest of the algorithm, Cert. 27 is used instead of Cert. 12, and Cert. 28 is used instead of Cert. 7. Using this technique, other nodes than N_1 do not see U's key at all. The only identity information they can infer is that the user who effectively owns the computation is some user whom PA_i has directly or indirectly delegated the permission to access the resource R.

To further strengthen privacy, PA_i may encrypt parts of the certificates that it issues. Since these certificates will be used by PA_i itself for creating CRCs only, nobody else but PA_i itself needs to be able to decrypt the encryption. This makes it virtually impossible to find out the identities of the users that PA_i has issued rights in the first place.

6 Implementing the architecture

We are building a JDK 1.2 based prototype, where distinct JVM protection domains could delegate Java Permission objects, in the form of SPKI certificates, between each other. At this writing (September 1998), we have completed the integration of SPKI certificates to the basic JVM security policy system [25], implemented the basic functionality of ECDSA in pure Java [15], and integrated these two together so that the SPKI certificates are signed with ECDSA signatures, yielding improved performance in key generation.

Our next steps include facilities for transferring SPKI certificates between the Java Virtual Machines, and extending the Java security policy objects to recognize and support dynamically

created delegations. Initially, we plan to share certificates through the file system between a number of JVMs running as separate processes under the UNIX operating system.

In addition, we are building a prototype of the ISAKMP [18] security protocol framework. This will allow us to create secure connections between network separated JVMs. The ISAKMP also allows us to easily transfer SPKI certificates and certificate chains between the virtual machines.

In order to support dynamic search and resolving of distributedly created SPKI certificate chains [3], we are integrating the Internet Domain Name System (DNS) certificate resource record (RR) format into our framework. This will allow us to store and retrieve long living SPKI certificates in the DNS system [22].

7 Conclusions

In this paper we have shown how authorisation certificates combined with relatively fast, elliptic curve based public key cryptography can be used to dynamically delegate authority in a distributed system. We analyzed the trust requirements of such a system in a fairly generic setting (Sect. 5.1), illustrated the details of how these trust requirements can be represented and verified with SPKI certificates (Sect. 5.2), and explained how the agents delegate permissions at run time by creating new key pairs and certificates. Finally, we outlined how the system can be utilized in a way that the user's identity is kept anonymous while still keeping all authorisations and connections secure (Sect. 5.4).

We are in the process of implementing a prototype of the proposed system. At the moment, we have completed the basic integration of SPKI certificates into the JDK 1.2 access control system (Sect. 3) and our first pure Java implementation of the ECDSA algorithms (Sect. 4). The next step is to integrate these with a fully distributed certificate management and retrieval system. The resulting system will allow distributed management of distributed systems security policies in fairly generic settings. In our view, the system could be used, e.g., as an Internet wide, organization borders crossing security policy management system.

References

1. Amoroso, E., *Fundamentals of Computer Security Technology*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.
2. Arnold, K. and Gosling, J., *The Java Programming Language*, Addison-Wesley, 1996.
3. Aura, T. , "Comparison of Graph-Search Algorithms for Authorisation Verification in Delegation", *Proceedings of the 2nd Nordic Workshop on Secure Computer Systems*, Helsinki, 1997.
4. Beth, T., Borcherding, M., Klein, B., *Valuation of Trust in Open Networks*, University of Karlsruhe, 1994.
5. Blaze, M., Feigmenbaum, J., and Lacy, J., "Decentralized trust management", *Proceedings of the 1996 IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, CA, May 1996.
6. Chadwick, D., Young, A., "Merging and Extending the PGP and PEM Trust Models - The ICE-TEL Trust Model", *IEEE Network Magazine*, May/June, 1997.
7. Ellison, C. M., Frantz, B., Lampson, B., Rivest, R., Thomas, B. M. and Ylönen, T., *Simple Public Key Certificate*, Internet-Draft `draft-ietf-spki-cert-structure-05.txt`, work in progress, Internet Engineering Task Force, March 1998.
8. Ellison, C. M., Frantz, B., Lampson, B., Rivest, R., Thomas, B. M. and Ylönen, T., *SPKI Certificate Theory*, Internet-Draft `draft-ietf-spki-cert-theory-02.txt`, work in progress, Internet Engineering Task Force, March 1998.
9. Ellison, C. M., Frantz, B., Lampson, B., Rivest, R., Thomas, B. M. and Ylönen, T., *SPKI Examples*, Internet-Draft `draft-ietf-spki-cert-examples-01.txt`, work in progress, Internet Engineering Task Force, March 1998.
10. Ellison, C., "Establishing Identity Without Certification Authorities", In Proceedings of the *USENIX Security Symposium*, 1996.

11. Gong, Li, *Java™ Security Architecture (JDK 1.2)*, DRAFT DOCUMENT (Revision 0.8), `http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.html`, Sun Microsystems, March 1998.
12. Gong, Li and Schemers, R., "Implementing Protection Domains in the Java Development Kit 1.2", *Proceedings of the 1998 Network and Distributed System Security Symposium*, San Diego, CA, March 11–13 1998, Internet Society, Reston, VA, March 1998.
13. International Telegraph and Telephone Consultative Committee (CCITT): *Recommendation X.509, The Directory - Authentication Framework*, CCITT Blue Book, Vol. VIII.8, pp. 48-81, 1988.
14. Kohl, J. and Neuman, C., *The Kerberos Network Authentication Service (V5)*, RFC1510, Internet Engineering Task Force, 1993.
15. Kortesniemi, Y., "Implementing Elliptic Curve Cryptosystems in Java 1.2", in *Proceedings of NordSec'98*, 6-7 November 1998, Trondheim, Norway, November 1998.
16. Landau, C., Security in a Secure Capability-Based System, *Operating Systems Review*, pp. 2-4, October 1989.
17. Lehti, I. and Nikander, P., "Certifying trust", *Proceedings of the Practice and Theory in Public Key Cryptography (PKC) '98*, Yokohama, Japan, Springer-Verlag, February 1998.
18. Maughan, D., Schertler, M., Schneider, M. and Turner, J., *Internet Security Association and Key Management Protocol (ISAKMP)*, Internet-Draft `draft-ietf-ipsec-isakmp-10.txt`, work in progress, Internet Engineering Task Force, July 1998.
19. McMahon, P.V., "SESAME V2 Public Key and Authorisation Extensions to Kerberos", in *Proceedings of 1995 Network and Distributed Systems Security*, February 16-17, 1995, San Diego, California, Internet Society 1995.
20. Nikander, P. and Karila, A., "A Java Beans Component Architecture for Cryptographic Protocols", *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, Usenix Association, 26-29 January 1998.
21. Nikander, P. and Partanen, J., "Distributed Policy Management for JDK 1.2", In *Proceedings of the 1999 Network and Distributed Systems Security Symposium*, 3-5 February 1999, San Diego, California, Internet Society, February 1999.
22. Nikander, P. and Viljanen, L., "Storing and Retrieving Internet Certificates", in *Proceedings of NordSec'98*, 6-7 November 1998, Trondheim, Norway, November 1998.
23. OMG, *CORBAservices: Common Object Services Specification, Revised Edition*, Object Management Group, Farmingham, MA, March 1997.
24. Partanen, J. and Nikander, P., "Adding SPKI certificates to JDK 1.2", in *Proceedings of NordSec'98*, 6-7 November 1998, Trondheim, Norway, November 1998.
25. Partanen, J., *Using SPKI certificates for Access Control in Java 1.2*, Master's Thesis, Helsinki University of Technology, August 1998.
26. Rivest, R. L. and Lampson, B., "SDSI — a simple distributed security infrastructure", *Proceedings of the 1996 Usenix Security Symposium*, 1996.
27. Wilhelm, G. U., Staamann, S., Buttyán, L., "On the Problem of Trust in Mobile Agent Systems", In *Proceedings of the 1998 Network And Distributed System Security Symposium*, March 11-13, 1998, San Diego, California, Internet Society, 1998.
28. Yahalom, R., Klein, B., Beth, T., "Trust Relationships in Secure Systems - A Distributed Authentication Perspective", In *Proceedings of the IEEE Conference on Research in Security and Privacy*, 1993.