PERFORMANCE ANALYSIS OF A SECURITY ARCHITECTURE FOR ACTIVE NETWORKS IN JAVA

Bernardo Alarcos[1], Enrique de la Hoz[2], Marifeli Sedano[3], María Calderón[4]
[1,2] Engineering Telematic Area, Alcalá University, Crta. Madrid-Barcelona, Km 33.600, 28871, Alcalá de Henares
[3] Department of Telematic Systems Engineering, UPM, ETSI Telecommunication, C. Universidad S/N, 28040, Madrid
[4] Department of Telematic Engineering, University Carlos III of Madrid, Av. Universidad 30, 28911, Leganés (Madrid), Spain
[1,2]{bernardo,enrique}@aut.uah.es, [3]marifeli@gsi.dit.upm.es, [4]maria@it.uc3m.es

Abstract

Active network technology supports the deployment and execution on the fly of new active services, without interrupting the network operation. Active networks are composed of special nodes (named Active Router) that are able to execute active code to offer the active services. This technology introduces some security threats that must be solved using a security architecture. We have developed a security architecture (ROSA) for an active network platform (SARA). Java has been used as programming language in order to provide portability, but it imposes some performance limitations. This paper analyses the penalty of using Java and proposes some mechanisms to improve the performance of cryptographic implementations in Java.

Key Words

Active networks, security, cryptography, Java, JNI.

1. Introduction[1]

There is a clear trend towards extending the set of functions that network routers support beyond the traditional forwarding service [1]. Active network [2] technology supports the deployment and execution on the fly of new active services, without interrupting the network operation. In this way, an active network is able to offer dynamically customized network services to customers/users. Potential advantages of active networking include the opening up of the network to third parties, the easy introduction of sophisticated and unanticipated network services, and rapid deployment of such services. Therefore, Active network technology enables fast deployment of new network services tailored to the specific needs of end users, among other features.

Some examples of applications that benefit from the active network are mobile IP, reliable multicast, web caching and transcoding of a data flow to adapt it to the link features.

An active network has some especial nodes named *Active routers*, which possess the capability of executing active code and, as a result, doing a tailored processing of active packets. However, the possibility of injecting active code into active routers introduces security risks that must be taken into account.

SARA [3] (Simple Active Router Assistant) is an active router prototype developed by Carlos III University of Madrid[2] in the context of the IST project GCAP[3]. It has been proposed a business model for SARA and its security risks have been analyzed. As a result a security architecture named ROSA (Realistic Open Security Architecture for Active Networks) has been proposed [4][5].

Current distributions of SARA prototype are implemented in Java language, so ideally it can run over any platform. Therefore, ROSA is implemented in Java language in order to integrate it into SARA. However, the cryptographic processes require intensive use of processor, what provokes considerable time consumption. When Java is used this time is bigger than when other programming languages such as C are chosen, and of course, bigger than when cryptographic hardware is used.

We have analysed some mechanisms to improve the performance of the security implementation. In order to do so, we propose some improvements in the security implementation: the initiation of cryptographic procedures, caching of the session key, the use of faster cryptographic providers and the use of JNI (Java Native Interface) cryptographic implementation. These mechanisms can be generalized to any cryptographic implementation in Java.

Henceforth, we briefly introduce SARA (section 2) and the model used to give active services (section 3). Then, section 4 is devoted to security threats in the proposed model and the security solution, ROSA. After that, in section 5, we explain how to provide the confidentiality of active packet service by using ROSA. Later, in section 6 we present some mechanisms to improve the performance

[1] This work has been funded by CICYT under project AURAS.

[2] SARA home site. http://matrix.it.uc3m.es/~sara

[3] GCAP IST project home page. http://www.laas.fr/GCAP

of ROSA implementation. Finally, section 7 is dedicated to conclusions.

2. Active networks and SARA platform

Two different approaches may be used to support dynamic network programmability: discrete approach and integrated approach.

The discrete approach means that packets do not include the code to be executed in the active routers, but there is a separate mechanism to inject programs into an active router. Frequently this download is done from a code server or other system with the responsibility of storing the code.

The integrated approach denotes that active packets (called capsules) not only include user data but they also include the code to process the own packet as well. This code is then executed at the active routers, when the active packets are propagated over the network.

SARA follows the discrete approach and is based on the router-assistant paradigm. It means that active code does not run directly on the router processor but on a different device, called assistant, which is directly attached to the router through a high-speed LAN. Hence, the router only has to identify and divert active packets to its assistant. Active packets are identified by the router alert option. This enables active node location transparency, since active packets need not to be addressed to the active router for this router to process them. After the assistant performs the requested processing, packets are returned to the router in order to be forwarded. The active code needed to process active packets is dynamically downloaded from Code Servers when it is not locally available at the assistant. In this way, safety is checked in advance, since only registered harmless-proofed code is allowed to run on the network. Thus the presumed target scenario is one where a central administrator provides active services loaded on the fly from a choice of known applications that have been supplied by the customer or network manager.

SARA is available in two platforms: One is fully based on Linux (playing roles, router and assistant as a development scenario); the other one is a hybrid platform where the router used is an Ericsson-Telebit AXI462 which runs a kernel adapted to work with an active assistant.

3. Packet Exchange process using SARA

In order to present the security architecture, we first introduce the packet exchange process. In this way, we can detect the requirements imposed by security and scalability concerns.

The elements involved in the packet exchange are:

- *Source* is the user terminal that generates active packets, and sends them towards the *Destination*.
- *Destination* is the terminal that receives the active packets sent by the *Source*.
- *Active Router* is a router capable of processing active packets. It is also able to obtain the active code needed.
- *Code Server* is the active code repository that serves the *Active Routers*.

The packet exchange description depicted in figure 1 is described next. Sometimes the *Source* needs special active processing for a flow of packets between the *Source* itself and *Destination.* In that case, the *Source* must send active packets (AP message in figure 1), addressed to the *Destination*. In those packets the *Source* indicates the identification of the active code to be executed. When an active packet reaches an *Active Router*, it is examined and the identification of the active code is extracted. If the active code is locally available at the *Active Router*, it performs the requested process and then forwards the packet towards the *Destination*. If the active code needed is not locally available, the *Active Router* requests it from the *Code Server* (Cr message). The *Code Server* then sends the requested code to the *Active Router* (Cd message), which now processes the packet and forwards it to the next hop. All the *Active Routers* along the path execute the same procedure until the packet reaches the *Destination*. The following active packets will presumably follow the same path, so the *Active Routers* will be capable of processing them without the need of requesting the code from *Code Servers* again.

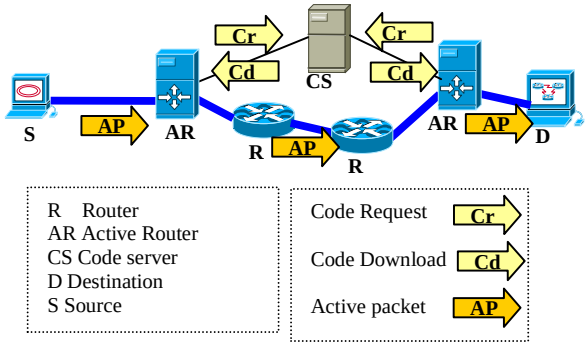

Figure 1. Packet exchange procedure

4. ROSA specification

Active networks impose security requirements that have been detailed in several documents [6]. Afterwards we present a summarized analysis of security threats that have been taken into account in the security solution. This analysis is made from the point of view of the different components of the scenario depicted in last section:

- Source/User (we consider that there is a user in the Source that requests active services):
  - Users must be capable of requesting authorization to the active network to send active packets that must be processed by the Active Router.
  - A user requesting an active service must provide authentication and non-repudiation. The last requirement is especially important when active services are provided in a commercial fashion.

Thanks to the providing of non-repudiation, we can consider the legal issue requirements.

- Users expect that the active network is processing their active packets only using reliable active code.
- Source must be capable of being sure that only the authorized user is requesting active services on its behalf.
- Source must be the only one capable of controlling its active services. This means that no other user is able to insert new active packets or to modify those active packets sent by Source, what would interfere with the requested active service.

- Active Router must verify:
  - The user that is sending the active packets is authorized to execute the active code at that moment.
  - Active code comes from an authorized Code Server.
  - Active code has not been maliciously modified.
- Code Server must be able to:
  - Authenticate Active Routers that are requesting active code, since not all the code will be available to all routers.
  - Send active code to authorized Active Routers with integrity and confidentiality protection, in order to prevent unauthorized parties from inspecting or modifying the delivered code.

Destination does not impose requirements since it does not demand active services from the network. It should be noticed that end-to-end security is out of the scope of this security solution.

4.1. Security Architecture

We have seen in the previous section, that user must request active service to obtain authorization from the active network. This requirement introduces a new component in the active network architecture, named *Authorization Server* (*AS* in figure 2).

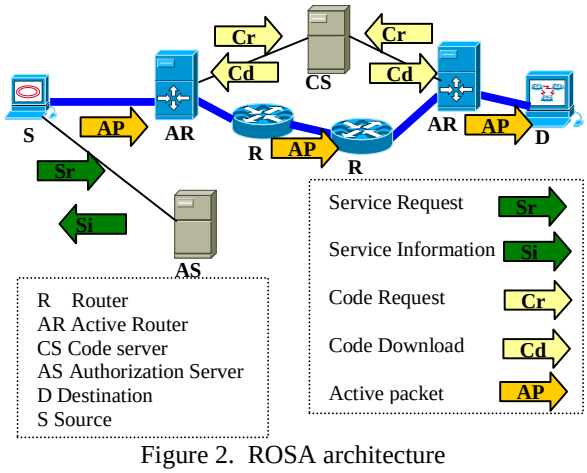

Figure 2. ROSA architecture

This new component has authorization information about the users. Therefore, authentication and authorization may be performed when *AS* receives a service request from a *Source*. Therefore, ROSA is applied in a scenario (figure 2) composed of the SARA components, *Active Router (AR)* and *Code Server* (*CS*), the ROSA component *AS*, and the end systems (*Source* and *Destination*).

4.2. Security mechanisms

The following processes can be distinguished in the security solution: service request, session key generation, active packet processing, and code download. Afterwards, we present the main mechanisms used in this processes. In [5] it is shown a more detailed justification for the criteria followed to select these mechanisms instead of others.

4.2.1. Service request and session key generation

A service request message is sent from the *Source* to the *AS*. This message must be signed using the user private key in order to offer authentication and non-repudiation.

The user makes use of this request to specify the parameters that identify an active session: User identifier (U), Service Start Time (SST) end Service End Time (SET), the code (Ci), which identifies the active code that must process the active packets in the active routers, and the IP addresses of Source (IP_S) and Destination (IP_D).

These parameters are used by *AS* to verify authorization of authenticated user. If the verification of authorization is right, *AS* accepts the session and gives a session key K to the user. In the following section we explain how the Source uses K to protect the active packets.

The Session Key is generated by the *AS* using a Key Derivation Function (KDF) and the following parameters:

$$K=KDF(U, SST, SET, IP_S, IP_D, Ci, Kci)$$

Where Kci is the secret associated to the active code i. Kci is shared by *AS* and *CS*. In order to refresh the Kci secrets, new secrets for all active codes are periodically generated by the *AS* and sent to the *CS* via a confidential channel. In the next section we see that *CS* distributes the Kci to the *Active Routers*.

The communication between *Source* and *AS* is performed over a TLS session. This kind of session offers mutual authentication services and confidentiality for the packets transferred in both directions.

4.2.2. Packet protection and verification

Once *Source* has requested the service and has received K from *AS*, it generates active packets towards the destination within the active session. In other words, active packets get the authorization to be sent from the *Source* S to the *Destination* D within the period of time between SET and SST. They are sent by the user U (which knows K) and with the code identifier Ci. To do this, *Source* protects active packets using a MAC (Message Authentication Code), in order to offer authentication and integrity services. MAC is generated using the hmac mechanism and K.

$$MAC = hmac (K, active\ packet)$$

Two different hmac algorithms are supported by the current version of ROSA, hmac-md5 and hmac-sha. A

field in the SARA header of the active packets is used to select the hmac algorithm used in that packet.

When an *Active Router* receives the active packet and the associated MAC, it must know K in order to verify authentication and integrity of the packet.

The AR generates K itself using the same KDF algorithm as the one used by the *AS*. To do this, the *AR* must know the values of U, SST, SET, IP_S, IP_D, Ci and Kci. The secret Kci is downloaded from the *CS* and the rest of parameters, are sent inside the active packet. Remember that *CS* shares the secret Kci with *AS*.

This solution uses hmac to protect active packets. This is an efficient algorithm, based on symmetric key. But symmetric key systems are usually associated with the key distribution problem. However, in this case we have seen that the key distribution mechanism is done in a simpler and more efficient way.

4.2.3. Code and Kci downloading from Code Server

When an *AR* receives an active packet, it first verifies that the message is not obsolete, i.e. it is within the validity period (between SST and SET), and then it verifies the availability of the requested active code. When the code (and Kci) is not locally available, the *AR* downloads it, using a secure (TLS) connection, from the *CS* (Cr and Cd in figure 2). After that, the *AR* generates K, using U, SST, SET, IP_S, IP_D, the Ci extracted from the active packet, and the Kci obtained from the *CS* when the code was downloaded. If the HMAC is validated, it means that *User* is authorized to execute the requested code, so the *AR* processes the packet using the requested code and forwards it to the next hop. If any unauthorized user changes the session parameters (U, SST, SET, IP_S, IP_D, Ci) of the packet, the *AR* generates a different K and the verification of HMAC fails. In this way, these parameters act as authorization information that is carried in the active packets. As a result, the active packet is acting as a credential.

The same procedure is repeated on every *AR* along the path until the packet reaches the Destination. The subsequent active packets of the flow benefit from cached copies of the active code and Kci in every *AR*. The TLS connection provides mutual authentication between *CS* and *AR* and confidentiality for the active code and Kci.

4.3. Complete and summarized process

Next, we summarize the complete process, illustrated in figure 2. First, the *Source* requests authorization to the *AS* (Sr message) to execute an active code Ci in the network (getting an active service). Then, after receiving and verifying the request, the *AS* generates K as we explained in section 3.2.1. After that, the *AS* sends an encrypted message (Si message), which contains K, to the *Source*. The *Source* decrypts the message and obtains K. Then, the *Source* generates the active packets (AP message), which include the authorization parameters (U, SST, SET, IP_S, IP_D, Ci) and the MAC generated using K.

When an *AR* receives the active packet, it first verifies that the message is within the validity period (SST, SET), and then it verifies the requested active code availability. In case the code and/or Kci are not locally available, the *AR* downloads them (Cr and Cd messages). Afterwards, the *AR* generates K and verifies the integrity and authenticity of the packet. The same procedure is repeated on every *AR* along the path until the packet reaches the *Destination*. The subsequent active packets of the flow benefit from cached copies of the active code and Kci in every *AR*.

Notice that the solution presented is limited to one security domain, i.e. one *AS* providing keys. It is possible to extend the solution to multiple domains, but this is more than a trivial task and it will be presented in future works.

5. Confidentiality of active payload

The ROSA specification does not take into account the fact that active packets could be confidential since it is not a general requirement. However, some active applications might require confidentiality in the payload carried by the active packets. This means that the payload of active packets might have to be encrypted in the *Source* and decrypted in the *Destination*. This new feature involves minimal changes in ROSA implementation.

Now, two keys are needed; the session key, used to generate the hmac K_{hmac}, and the session key, used to encrypt $K_{encrypt}$. When the *User* requests an active service to the *Authorization Server*, the cryptographic procedure must be specified: authentication or encryption and authentication. The *Authorization Server* uses the KDF function to generate 1) K_{hmac} or, 2) K_{hmac} and $K_{encrypt}$ respectively:

1) K_{hmac}=KDF (U, SST, SET, IP_S, IP_D, Ci, Kci)

2) {K_{hmac}, $K_{encrypt}$}=KDF (U, SST, SET, IP_S, IP_D, Ci, Kci)

The KDF is the same function in both cases as it is usually a hash or hmac algorithm based function. The number of iterations of the KDF function determines the number of random bits generated to get one or two keys.

Next, *Source* sends active packets to the *Destination* with the payload encrypted, and with the authenticator generated by hmac.

When an active packet arrives to an *Active Router*, it knows if the payload is encrypted looking at the SARA header of the packet. If the payload is ciphered, *Active Router* generates both, K_{hmac} and $K_{encrypt}$, using the same KDF function and the same parameters as the ones used by the *Authorization Server*. Then the *Active Router* verifies authentication of the packet using hmac and K_{hmac}. In addition, if the payload is encrypted and it has to be read or modified by the Active router, it decrypts this payload using the encryption algorithm and $K_{encrypt}$.

6. ROSA Implementation

SARA is implemented over the Java Virtual Machine of Sun. As a result, the current version of SARA uses Sun

J2SDK 1.4.1 on Linux. Furthermore, the current version of the ROSA prototype uses Java language.

Three main capabilities had to be implemented in *Active Routers* supporting ROSA. First, the capability of generating HMAC (either SHA-1 or MD5), used to protect packets or to verify the packets that are crossing the router, and to generate the session key. Second, establishing and managing TLS connections in order to download code and keys. And finally, the third capability is decrypting and encrypting confidential payload (when using this service).

Now we will present three mechanisms that allow us to improve the performance of the Java based implementation.

6.1. Initializing cryptographic algorithms

In the early stages of ROSA development we noticed that cryptographic algorithms have very high initialization times. This is related to internal implementation issues of Java cryptographic classes and it is present in all the cryptographic providers we have tested. This initialization time only appears the first time that a cryptographic object (of any kind) is instantiated in our Java programs. Furthermore, if the cryptographic object is destroyed and then it is instantiated again, the delay will appear once more. For instance, if a secure application uses three different types of cryptographic objects it will suffer three expensive initialization processes (much longer even than any computation made by that object). This delay appears the first time that a cryptographic object is called by an active application. Moreover, the instance of a cryptographic object is associated to its application and is destroyed when the application ends, so the delay appears for every application that calls the cryptographic object. If we think of an environment with many secure applications being continuously loaded and using cryptographic algorithms, we can imagine that the delay may become quite noticeable.

In order to avoid that undesirable effect, we provide a mechanism that detects if security is supported by the active network platform. If so, the execution environment of active applications creates instances of the cryptographic objects that will be used, at start-up time. Therefore, the expensive delay of initializing cryptographic processes is only suffered once when the platform is booted. Furthermore, the instances are associated to execution environments and not to the applications, so they keep alive permanently.

We have seen that the time spends in the hmac/hash process the first time is 60/10 times bigger than the time spends in the same process the following times. So initializing the hmac and hash process in ROSA, we avoided the degradation of the performance the first time that we use the algorithm.

6.2. Caching session key

In order to verify every packet crossing the *Active Router*, we have to determine the proper session key for each packet. All the packets belonging to the same flow are authenticated (and must be verified) using the same session key. Therefore, we could use a cache for storing the already-computed session keys. This would turn the hmac (remember we use hmac as KDF) computation into a cache search, which is faster. Session key caching has been confirmed as a good performance improvement in the early development stages.

We have measured the times of the process inside a single application (*HelloWorld*). The percentage of time spends in every process is: key generation 12.5%, hmac verifying and generation 75%, rest of packet processing 12.5%. So we can save about 12.5% of time avoiding session key generation by using cache of session key.

6.3. Choosing a cryptographic provider

Due to its use of cryptographic services, ROSA makes use of Java Cryptography Extension JCE [7]. JCE provides a framework and implementations for encryption, key generation, key agreement, and MAC algorithms. JCE is based on the same design principles found in the Java Cryptography Architecture (JCA) [8]: implementation independent and, whenever possible, algorithm independent. JCE uses the same provider architecture. This means that we can plug providers signed by a reliable entity into the JCE framework adding seamlessly new implementations or new algorithms when desired. This "provider" architecture allows setting dynamically in runtime the algorithm implementation to use in each case. This allows selecting the fastest implementation of every algorithm in order to improve the platform performance just making minor changes to the existing code. Our goal will be, then, showing which of the freely available providers offers the best performance in each of the cryptographic services (MAC, Ciphers) needed. Moreover, we have built our own provider using the Java Native Interface (JNI) [9]. This provider only includes the algorithms needed. Theoretically, it should offer us very good performance. However, a JNI implementation introduces a disadvantage to be considered. SARA portability is reduced.

Next we present providers comparison. We have focused our attention on optimizing the mechanisms of hmac and encryption, which are the most frequently used by the active router. We have selected four Java cryptographic providers that fulfill the features to provide free and reliable Java cryptographic libraries:

- SunJCE (http://java.sun.com).
- Cryptix (http://www.cryptix.org).
- BouncyCastle (BC) (http://www.bouncycastle.org).
- FlexiProvider (http://www.flexiprovider.de).

We have compared the results of Java providers with a JNI implementation using the library libgcrypt[4], which is the GNU cryptographic library implemented in C language. Notice that some of the algorithms are not supported for all the providers.

[4] http://www.gnu.org/directory/security/libgcrypt.html

Then we have measured the delay introduced by these providers for each security procedure: hmac and encryption. Tests have been done using a PIII 1.1 GHz, 256MB, Linux Kernel 2.4.18.

We compare JNI implementation and JAVA providers of hmac-sha1 (figure 3) and hmac-md5 (figure 4).

The tests have been realized measuring the delay introduced by every provider for different packet sizes (between 256 bytes and 2048 bytes).

The results obtained show that the fastest Java provider for hmac-md5 is BC and SunJCE is the fastest for hmac-sha1. Furthermore, we can see that the best Java provider implementation is around three times (300%) slower than the JNI implementation.

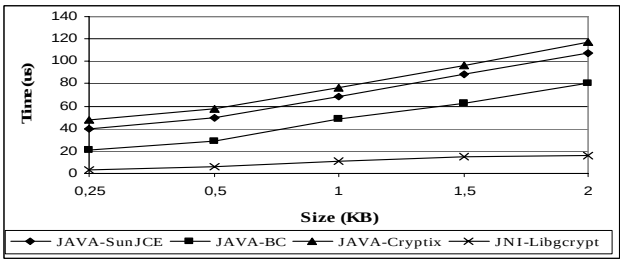

Figure 3. HMAC-MD5

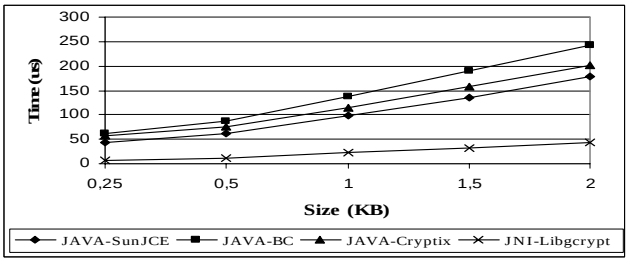

Figure 4. HMAC-SHA1

Next we have compared the symmetric encryption algorithm AES for different Java cryptographic providers with the JNI implementation. The results obtained show that (figure 5) BC is the fastest Java provider and JNI implementation is around three times (300%) faster.

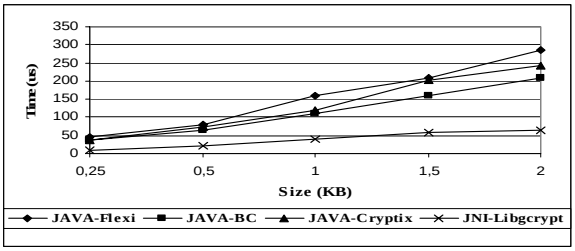

Figure 5. AES encryption

JNI provides better performance than Java providers but SARA's portability is reduced. Due to this, it must be included in SARA the capability of configuring the providers (e.g. using a XML configuration file). Such providers can be Java security providers or our JNI implementation. Therefore, the active router administrator must select the cryptographic modules among those distributed by SARA. As a result, if the administrator knows that his platform supports the JNI modules, he will select them to improve the performance.

7. Conclusion

Some mechanisms can be used to improve the performance of a Java security implementation (as ROSA): initializing cryptographic process in the execution environment at start-time, caching frequently used session keys, and selecting the fastest Java cryptographic providers or JNI implementation.

We have compared the delays introduced by some Java providers in order to select the fastest. From the results of our comparison we conclude that we may exploit JCE ability of using different cryptographic providers, to select the fastest provider for every algorithm. In this case we select SunJCE for hmac-sha1 and BC for hmac-md5 and AES encryption.

A better improvement consists on using a JNI implementation. The results obtained show that JNI implementations introduce an improvement of around 300% over the best Java provider. But SARA's portability is reduced in this case. Thereby, we have proposed to enable the administrator to configure the use of Java or JNI cryptographic procedures in SARA implementation.

References

[1] Wetherall, D. J., Legedza, U., Guttag, J. Introducing new Internet services: Why and How. *IEEE Network Magazine,12*(3), 1998, 12-19.

[2**]** Tennenhouse, D. L., Wetherall, D. J. Towards an Active Network Architecture. *Computer Communication Review*, *26*(2), 1996, 5-18.

[3**]** Larrabeiti, D., Calderón, M., Azcorra, A., Urueña, M. A practical approach to network-based processing, *Proc. 4th International Workshop on Active Middleware Services*, Edinburgh, Scotland, 2002.

[4**]** M. Bagnulo, B. Alarcos, M. Calderón, M. Sedano, Providing Authentication & Authorization mechanisms for active service charging, *International workshop on Internet Charging and QoS Technologies ICQT'2002*, Zurich, Switzerland, 2002, 337-346.

[5**]** M. Bagnulo, B. Alarcos, M. Calderón, M. Sedano, ROSA: Realistic Open Security Architecture for active networks, *Proc. 4th International Working Conference IWAN'2002*, Zurich, Switzerland, 2002, 204-215.

[6] AN Security Working Group, Security Architecture for Active Nets, 2001.

[7] Java Cryptography Extension (JCE) Reference Guide http://java.sun.com/j2se/1.4/docs/guide/security/jce/JCERefGuide.html

[8] Java Cryptography Architecture API http://java.sun.com/j2se/1.4.1/docs/guide/security/CryptoSpec.html

[9] Java Native Interface Specification http://java.sun.com/j2se/1.4.1/docs/guide/jni/spec/jniTOC.doc.html