Encrypting Java Archives and its Application to Mobile Agent Security

Volker Roth[1] and Vania Conan[2]

[1] Fraunhofer Institut für Graphische Datenverarbeitung
Rundeturmstraße 6, 64283 Darmstadt, Germany
vroth@igd.fhg.de
[2] Thomson-CSF Communications
66, Rue du Fossé Blanc
BP82, 92231 Gennevilliers Cedex, France
Vania.Conan@tcc.thomson-csf.com

Abstract. In this article we describe an extension of Java Archives that allows to keep data encrypted for multiple recipients. Encrypted data is accessible only by selected access groups. Java archives may be used as containers of mobile agents, which allows agents to keep confidential data unaccessible while residing on untrusted hosts. However, additional protective measures are required in order to prevent Cut & Paste attacks on mobile agents by malicious hosts. One such mechanism is described. The usefulness of the concepts is illustrated by an example application for user profile management in an electronic commerce setting.

Keywords: mobile agent security, Java Archives, encryption, malicious hosts

1 Motivation

Mobile agents [22] push the flexibility of distributed systems to their limits since not only computations are distributed dynamically, the code that performs them is also distributed. A number of mobile agent systems are in existence at present; basic information on about 60 such systems was collected[1] in the run-up to the ASA/MA'99 Conference that took place at the beginning of October in Palm Springs, FL, USA.

Adequate security has been identified numerous times by different researches as a top criterion for the acceptance of mobile agent technology.

[1] See URL <http://www.informatik.uni-stuttgart.de/ ipvr/ vs/ projekte/ mole/ mal/ mal.html >

Despite advances in conceptual mobile agent security issues [19, 20], few agent systems actually seem to offer security mechanisms beyond transport layer security. In mobile agent systems,

1. agents must be protected against malicious hosts,
2. hosts must be protected against malicious agents,
3. agents must be protected against other agents,
4. both agents and hosts must be protected against the rest of the world.

The problem of malicious hosts is generally agreed to be the most challenging one of those noted above. A number of protection schemes have been devised to protect certain aspects of mobile agents against malicious hosts, yet most of them are very restricted or fail to be applicable in a general setting. In this article we describe another mechanism falling into the category "restricted but applicable".

Our philosophy is that good servers should help good agents to protect themselves against bad servers. In particular, mobile agent servers should offer some transparent security services to agents. This also has the advantage that bad servers can not make agents "forget" to take care of their security on their subsequent hops.

Once a mobile agent leaves the trusted haven of its owner's computer and hops off to another host, it is more or less on its own (unless it co-operates with other agents in protecting mutual security objectives [12]) and at the mercy of its hosting server. Even though an agent may compute certain functions in privacy while being on an untrusted host [16], the general rule holds: If data needs to be confidential then it must be encrypted and the encryption key must be unavailable even to the agent itself. Decryption, and hence disclosure of the encryption key, may be delayed until the very last moment [10] but ultimately the host may learn it when the data is being used.

However, if data of an agent need only be accessed on particular hosts then it may be encrypted in such a way that it is unavailable to other hosts when the agent passes by them. Contemporary agent systems occasionally advertise encryption of agents as a security feature but in general this means encryption of agents that are in transit between host systems (for instance using SSL). While this is the state-of-the-art protection mechanism against network-based eavesdroppers this does not prevent individual hosts from spying on agents at all. We are aware of only one

mobile agent system which supports partial encryption of agent data for particular recipient servers. This agent system is called *Ajanta* [7, 8].

Ajanta provides a number of mechanisms which are comparable to the ones we describe in this article. Agents can have a *read-only* state which is protected by means of digital signatures. A *targeted state* is used to reveal parts of the agent's state to selected recipients. However, Ajanta appears to be vulnerable to *cut & paste* attacks on the targeted state. This type of attack is described in Section 3.

In this article, we describe an addition to the Java Archive (JAR) Format that supports transparent selective encryption of archive contents for multiple recipients (Section 2). In Section 3, we describe the use of such JARs as containers for mobile agents (initial work on this subject is described in [13]), and show how *cut & paste attacks* on encrypted contents can be prevented.

2 Java Archive Extensions

The Java Archive (JAR) Format [17] devised by Sun Microsystems builds on the popular ZIP archive format. It supports multiple signatures and multiple signers on subsets of a JAR's contents. Multiple signatures are computed and verified efficiently through a two-stage process that avoids re-hashing of JAR contents for each signature. JARs are frequently used to distribute Java class packages, Applets and Java Beans. The original JAR Format devised by Sun does not provide mechanisms for encrypting parts of the archive; only digital signatures are covered. In this section we describe an extension to the Java Archive Structure that allows to keep partial archive data encrypted for multiple recipients.

The extension consists of an additional meta–folder with the name *SEAL-INF*. This folder stores the additional files required for:

- meta-information for the management of encryption and decryption (one file with the name *INSTALL.MF*).
- encrypted archives, identified by the extension *EAR*.
- data structures containing the encrypted data encrypted keys, identified by the extension *P7*.

The encryption mechanism used for encrypting EARs is a hybrid one. An encryption key is chosen randomly for a suitable symmetric bulk encryption cipher such as DESede/CBC/PKCS5Padding [4, 5, 15]. This key is

used to encrypt the compressed plain text. For each recipient, this *bulk encryption key* (BEK) is encrypted in the recipient's public key using an asymmetric cipher such as RSA [11]. This process is also called *sealing*, hence the name SEAL-INF for the meta-information folder. The sealing information and EARs are not located in the META-INF folder such that this information may itself be signed through the ordinary signing process defined in the JAR format. This is crucial for the application of Java Archives as containers of mobile agents as described in Section 3.

We chose PKCS#7 [14] as the standard for representing the encrypted BEK and recipient information. This blends well with the Java Archive Format, which mandates the use of PKCS#7 for DSA and MD5/RSA signatures. In choosing PKCS#7, we implicitly adopted X.509 [6] as the certificate format, which is the de-facto standard for the representation of certificates in the World Wide Web.

On creating a JAR, the user defines *access groups*, and assigns a name to each group. Each access group G with name $name_G$ consists of a set of valid recipients $r_1, \ldots, r_n$ and a randomly generated symmetric BEK k_G. Recipients are represented by their valid public key certificates. For each access group G, a PKCS#7 *EnvelopedData* structure is created, wrapped into a PKCS#7 *ContentInfo*, and stored in folder SEAL-INF under the name $name_G$.P7. This file contains a *RecipientInfo* for each intended recipient, holding k_G encrypted in the recipient's public key.

The user then may assign folders in the JAR to access groups. Each folder may be assigned to at most one such group. For each folder V that is assigned to an access group G, a corresponding entry is stored in the INSTALL.MF file, which is stored in the SEAL-INF folder of the JAR. This file is similar to MANIFEST.MF files in that it contains sections of name/value pairs formatted like header fields in RFC822 [2] messages. Each section is separated from its successor by an empty line. Each section contains the entries described below, quotation marks denote literal strings:

Name	Value
"Name"	V
"EAR"	$name_V$
"Group"	$name_G$

The unique EAR name $name_V$ is generated by the sealing software. The encryption process takes each folder V that is assigned to an access group

G, compresses its contents recursively into a ZIP archive, and encrypts it with k_G. The resulting file is stored in the SEAL-INF folder under the name $name_V$.EAR. The plain text folder V is then deleted.

The decryption process first tries to recover as many bulk encryption keys as possible by verifying the RecipientInfos in the P7 files against the public key certificates corresponding to the available private decryption keys. Each RecipientInfo contains the unique issuer name and serial number of the certificate [6] that was used to create that entry. This establishes the groups to which the processing entity belongs. For each folder that is assigned to such groups, the corresponding EAR is decrypted with the recovered BEK, and its contents are decompressed to folder V.

3 Encrypted JARs and Mobile Agents

A number of agent systems represent agents simply as a stream of serialised objects that encapsulate virtually all the information in the agent, including any data the agent may have collected on previous hops. Classes are either downloaded on demand or the serialised stream is annotated with the byte code. Often, RMI is used as means of transporting the agent from one hop to the next. While this bears advantages such as simplicity and elegance, it puts strains on security mechanisms. Since data and object state is cluttered throughout the serialised stream and many alternative orderings exist for a serialised object graph, it is hard to apply e.g. digital signatures to portions of an agent's data transparently for an agent.

Moreover, it is complicated to infer any information from the agent's representation before the agent is actually deserialised. This is unfortunate because during deserialisation the agent's classes are installed and the agent may seize control over the deserialisation thread by implementing the readObject and writeObject methods described in the documentation of class *ObjectInputStream*.

On the other hand, the JAR Format already offers well-defined processes for the signing and signature verification of archive contents. An agent's JAR may be loaded in its entirety and verified and/or processed in a number of ways transparent to the agent even before the agent is run. The basic layout that we use for such JARs is shown in Table 1. Once an agent is admitted to the system, its JAR is decompressed and installed in a file system folder that is reserved for that particular agent. The location

META-INF/	MANIFEST.MF
	alias.SF
	alias.(DSA|RSA|PGP)
SEAL-INF/	INSTALL.MF
	$name_V$.EAR
	$name_G$.P7
static/	agent.properties
mutable/	instance.ser
classes	

Table 1. The extended structure of a JAR used as a container for mobile agents

of this folder is passed to the mobile agent. The Agent is granted access to this folder and can use it as storage space for data that it acquires. We encourage agent programmers to use this feature because this reduces the amount of data that is occupied by agents in the memory of the server's VM. Moreover, it is persistent storage that is not lost in case of a server crash. On migration, the agent's folder is compressed into a JAR again.

However, the mechanisms described in Section 2 do not yet suffice to assure the protection of the encrypted data. Malicious hosts and other attackers that get a copy of the agent JAR may launch a *cut & paste attack*. The following example illustrates the attack:

1. Alice prepares a search agent. The agent collects stock quotes from Bob's server and the server of Mallet, but Alice does not want Mallet to know which quotes her agent collected from Bob. So she creates an access group G with Bob as its sole recipient and assigns the folder secret to this group. The agent is programmed to store the quotes in that folder if it is at Bob's server.
2. Alice sends her agent to Bob. Bob decrypts and installs the folder secret because he is a legal recipient. The agent collects the stock quotes and sets its next hop to the server of Mallet. Bob re-encrypts the folder and sends the agent to Mallet.
3. Mallet copies the INSTALL.MF, $name_G$.P7 and $name_V$.EAR files from the agent to an agent of its own and sends it to Bob.
4. Bob decrypts and installs the folder secret in Mallet's agent because he is a valid recipient. The agent then copies the plain text data to another folder and sets its next hop to Mallet. Bob re-encrypts folder secret and sends the agent back to Mallet.
5. Mallet reads the plain text returned by his agent.

The attack is successful because the encrypted archives are not linked to the agent instance and Alice. Signing the encrypted archive is of no help since the signature may simply be stripped away by Mallet. One way to forge such a link is to request a non-interactive proof of knowledge of k_G from the entity that claims to be the rightful owner of the agent. In addition to this, the agent must have a unique static kernel that can be signed by its owner as proof of ownership and authorisation. The information in the kernel must be sufficient in order to assure that the agent can be bootstrapped securely, e. g. by starting only a class that the agent's owner trusts to keep confidential information in the appropriate folders. Below, we describe an approach to create a safe link.

Let cert_A be the certificate of the signing key of Alice. Let MAC be a suitable *Message Authentication Code* (see [9], Section 9.5). Alice adds one additional section with the reserved name "GROUPS" to the INSTALL.MF file of the agent. For each defined access group G_i she puts an entry into this section as shown below; quotation marks denote literal strings:

Name	Value
"Name"	"GROUPS"
$name_{G_1}$	$\text{MAC}(k_{G_1}, \text{cert}_A)$
$name_{G_2}$	$\text{MAC}(k_{G_2}, \text{cert}_A)$
...	

Alice signs the static parts of her agent including the agent properties, the file INSTALL.MF, and the agent's classes with her secret signing key. The properties contain the agent's unique name, the name of its main class, and any other properties Alice wants to define in a way that cannot be tampered with without breaking the signature and hence Alice's assertion of ownership of her agent. Bob verifies the validity of access groups in Alice's agent as described in Algorithm 1.

Subsequent to this test, Bob iterates through the sections in file INSTALL.MF; for each folder V that is assigned to an accepted access group G Bob decrypts the appropriate $name_V$.EAR and installs it in folder V of the agent.

4 Security

The technical security of the encrypted data in the agent is based on the security of the weakest link in the chain of cryptographic primitives

Algorithm 1 The algorithm for verifying access group validity.

```
{ Let cert_B be the certificate of Bob's decryption key. }
{ Let cert_A be the certificate of Alice's signing key. }
for all name_G do
   Bob loads the EnvelopedData structure SEAL-INF/name_G.P7;
   if it contains a RecipientInfo matching cert_B then
      Bob recovers k_G with the private key corresponding to cert_B;
      Bob computes MAC(k_G, cert_A);
      Bob compares the result with the value of attribute name_G in section GROUPS;
      if both are equal then
         accept G;
      else
         reject G;
      end if
   end if
end for
```

consisting of a symmetric cipher, the signature scheme, the weakest asymmetric encryption used within a RecipientInfo, and the MAC algorithm.

The MAC is crucial for the prevention of cut & paste attacks. In order to launch a successful attack, Mallet has the following choices. He may:

- Convince Bob that he produced the EAR by forging a MAC with his own certificate as the input and without knowing k_G (otherwise Mallet may simply decrypt the cipher text).
- Impersonate Alice, which requires forging Alice's signature on the kernel of an agent of his own.
- Modify the state of Alice's agent such that it leaks the plain text data. Copying the P7 and EAR files to a different agent of Alice won't work because the INSTALL.MF file is covered by Alice's signature.

Even if Mallet convinces a certificate authority Bob trusts to issue a certificate with Alice's identity and Mallet's public key, this will be detected by Bob, because the MAC is computed by Alice on her original certificate, which includes her public key and which is used to verify her signature on her agent's kernel.

Mallet cannot substitute a class of his own as the principal agent class because the class and its name is covered by Alice's signature. However, he may modify the serialised instances of Alice's agent such that a Trojan horse class is called by it, which leaks the plain text data. Therefore, it is of utmost importance that access to the agent's folder is granted only

to classes that are authorised by Alice, using the Java 2 AccessController mechanisms and the digests stored in the Manifest file of the JAR.

5 Transparent Implementation

We integrated a reference implementation of the mechanisms described in Sections 2 and 3 into our experimental mobile agent server SeMoA. Transport of agents in SeMoA is handled by two principal services: the so-called *ingate* and *outgate*. Both make use of other services that may be registered in the server dynamically and at boot time. Services are grouped according to functionality and level of confidentiality on a number of configurable *service levels*. On such level is the *transport level* on which services are registered that have to do with transporting agents. A second level is the *security level* on which services are registered that provide security services. The ingate and outgate scan the security level for particular classes of services implementing *filters* for incoming and outgoing agents. They arrange such filters in a pipeline that must be passed by each agent before it is admitted to the server and before it is sent to its next hop. This is illustrated in Figure 1.

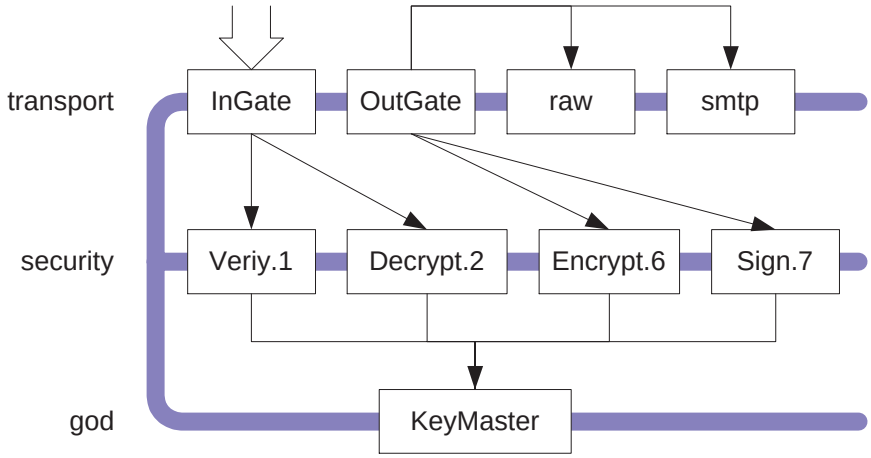

Fig. 1. An excerpt of the service levels in the experimental SeMoA server.

We implemented four security filters, two for incoming agents and two for outgoing agents:

Verify filter: This filter expects and verifies two signatures per agent. The first signature covers the static part of the agent, the signer is assumed to be its owner. The second signature covers the entire agent, the signer is assumed to be the last sender of the agent. The valid certificate chain of the signer's certificate must end in a trusted CA certificate.

Decrypt filter: This filter implements the decryption mechanisms described in Sections 2 and 3, including the verification of the access groups as set forth in Algorithm 1.

Encrypt filter: This filter re-encrypts the agent's contents according to the scheme described in Sections 2 and 3, and deletes the plain text folders.

Sign filter: This filter binds the new execution state of the agent to its kernel by signing the complete agent with the server's secret signing key.

Each server has two key pairs, one for signing and another one for encryption and decryption. The agent passes the incoming filter pipeline before it is started. On execution of the agent, the accessible data is already installed in the agent's folder. Apart from setting up the access groups and assigning the appropriate folders, the agent creator is not bothered with the encryption and decryption anymore. This is handled transparently by the server on behalf of the agent.

6 Application Example: Profile Protection

Privacy protection is an important feature for agent applications [1]. Mobile agents which carry personal information are able to carry out personalised tasks on their owner's behalf. The richer the profile information, the more personalised the agent's response. Securing the profile data is thus a means to insure privacy protection.

Privacy protection is guaranteed in the European Union by national laws, and national data protection organisations. All national regulations implement the same European privacy protection principles, as expressed in EU directive 95/46 [3], which is a legally binding document since October 1998. The directive proposes a formal framework for privacy protection, which is not available for instance in the USA. The directive considers personal data as information on which the data subject has a number of

rights such as right of access to the data, opt out opportunity and protection on international data flow. A detailed list is given in the directive.

For instance if a netizen (the data subject) provides his name and address (personal data) to the web site of a software vendor (the data controller), he automatically reserves said rights on this data, and the data controller implicitly agrees to adhere to these rights. In addition, the data controller has a number of obligations. Only legitimate data may be collected and the collected data must be adequate to the purpose for which it is collected.

Apart from legal issues involved in processing personal data, technical means must be provided to facilitate the management and control of such data in particular for the data subject. The W3C put forward a proposal named *Platform for Privacy Preferences* [21] (P3P) that aims at providing a protocol for reaching agreements between a Web user and a Web site on the exchange and use of personal data. P3P is reaching its final state in early 2000. Prototypes of P3P compliant servers and client applications are available, and are now ready for a widespread dissemination of the standard.

P3P consists of a negotiation and data exchange phase. The protocol is designed for a client server system. At first sight, mobile agents may not profit from a standard such as P3P since negotiating personal information while being disconnected from a trusted computing base is extremely risky in the face of a potentially malicious host.

However mobile agents can benefit from the on-line P3P negotiation phase. The agreement reached by the two parties, the data subject (netizen) and the data controller (retailer), is valid on a number of profile elements over a given period of time. An agreement id is stored on the netizen's side with the corresponding time stamp. Agreements with regular retailers may last up to six months or one year. Mobile agents can then carry the relevant agreement id and the corresponding profile elements using selective encryption as presented above. Only the authorised retailers will thus be allowed access to the information.

We are implementing this scheme for personalised product brokering mobile agents. Upon visiting the retailer's sites, answers to mobile agents are personalised, taking into account both the specific request and profile information.

7 Conclusions

In this article, we presented an extension of the JAR format that allows to encrypt contents in a JAR for multiple recipients, and its application to mobile agents. Using extended JARs as containers for mobile agents requires additional security precautions in order to detect and prevent cut & paste attacks on the encrypted contents. We presented an approach to solving this problem. In conjunction with the signing scheme we devised, we are now able to support a number of access rights to portions of an agent. Folders in the agent's structure may have one of the following access rights:

Read–only: This data can be read on each host but cannot be modified without breaking the agent's verifiable integrity.

Read/write committed: This data can be read and modified on each host but hosts have to commit to the new state. The changes can be (in principle) be checked and linked to that host on the agent's next hop.

Group read: This data can be read only on a predetermined set of authorised hosts. Modification of the data breaks the agent's verifiable integrity.

Group read/write: This data can be read and modified only on a predetermined set of authorised hosts.

Groups may be defined flexibly. The selective encryption scheme that we presented is highly useful to protect data a mobile agent gathers. Hosts not belonging to the access group of a given folder cannot eavesdrop on data in such folders. We realised a reference implementation of the encryption, decryption, signing, and verification steps including cut & paste detection and prevention; these operations are transparent for mobile agents. Hence, agents can remain completely unaware of the security operations performed on them as these operations are part of the agent server's security services.

For illustration, we described an application scenario that makes use of the selective encryption scheme for protecting personal information within mobile agents such that this information is made available only to the intended recipients.

References

1. Conan, V., Foss, M., Lenda, P., Louveaux, S., and SalaYn, A. Legal issues for personalised agent mediated electronic commerce: The aimedia case study. In *Agent Mediated Electronic Commerce*, C. Sierra, Ed. Springer Verlag, Berlin, 2000.
2. Crocker, D. H. Standard for the format of ARPA Internet text messages. Request for Comments 822, Internet Engineering Task Force, aug 1982.
3. EC. Directive 95/46/EC of the European Parliament and of the Council of 24 october 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data. published in OJEC, November 1995.
4. FIPS46. Data Encryption Standard. Federal Information Processing Standards Publication 46, U.S. Department of Commerce/National Bureau of Standards, National Technical Information Service, Springfield, Virginia, 1977. revised as FIPS 46-1:1988; FIPS 46-2:1993.
5. International Organization for Standardization. *Information Processing – Modes of Operation for an n–Bit Block Cipher Algorithm*. Geneva, Switzerland, 1991. ISO/IEC 10116.
6. International Organization for Standardization. *Information technology – Open Systems Interconnection – The Directory: Authentication Framework*. Geneva, Switzerland, nov 1993. ISO/IEC 9594-8, equivalent to ITU-T Rec. X.509, 1993.
7. Karnik, N. M., and Tripathi, A. R. Agent server architecture for the Ajanta mobile-agent system. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '98)* (Las Vegas, July 1998).
8. Karnik, N. M., and Tripathi, A. R. Security in the Ajanta mobile agent system. Technical Report TR-5-99, University of Minnesota, Minneapolis, MN 55455, U. S. A., May 1999.
9. Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. *Handbook of Applied Cryptography*. Discrete Mathematics and its Applications. CRC Press, New York, 1996. ISBN 0-8493-8523-7.
10. Riordan, J., and Schneier, B. Environmental key generation towards clueless agents. In Vigna [19], pp. 15–24.
11. Rivest, R. L., Shamir, A., and Adleman, L. M. A method for obtaining digital signatures and publi–key cryptosystems. *Communications of the ACM 21* (1978), 120–126.
12. Roth, V. Mutual protection of co-operating agents. In *Secure Internet Programming* [20].
13. Roth, V., and Jalali, M. Access Control and Key Management for Mobile Agents. *Computers & Graphics, Special Issue on Data Security in Image Communication and Networks 22*, 4 (1998), 457–461.
14. RSA Laboratories. Cryptographic message syntax standard. Public Key–Cryptography Standards 7, RSA Laboratories, Redwood City, CA, USA, 1993. Available at URL: `ftp://ftp.rsa.com/pub/pkcs/`.
15. RSA Laboratories. Password–based encryption standard. Public Key–Cryptography Standards 5, RSA Laboratories, Redwood City, CA, USA, 1993. Available at URL: `ftp://ftp.rsa.com/pub/pkcs/`.
16. Sander, T., and Tschudin, C. F. Protecting mobile agents against malicious hosts. In Vigna [19], pp. 44–60.

17. Sun Microsystems, Inc. *Java^{tm} Archive (JAR) Features*. in [18], relative URL: `file:/docs/guide/jar/index.html`.
18. Sun Microsystems, Inc. *JDK 1.2 Documentation*, 1998. Available at URL: `http://java.sun.com`.
19. Vigna, G., Ed. *Mobile Agents and Security*, vol. 1419 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin Heidelberg, 1998.
20. Vitek, J., and Jensen, C. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, vol. 1603 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1999.
21. W3C. Platform for Privacy Preferences (P3P) Specification. Available from URL `HTTP://www.w3.org/TR/1999/WD-P3P-19990826/`, August 1999.
22. White, J. E. *Mobile Agents*. AAAI/MIT Press, 1997, ch. 18.