Secure Mediation with Mobile Code*

Joachim Biskup, Barbara Sprick, and Lena Wiese

Universität Dortmund, D-44221 Dortmund, Germany
{biskup, sprick, wiese}@ls6.cs.uni-dortmund.de
http://ls6-www.cs.uni-dortmund.de/issi/

Abstract. A mediator helps a client of a distributed information system to acquire data without contacting each datasource. We show how mobile code can be used to ensure confidentiality of data in a secure mediation system. We analyze what advantages mobile code has over mobile data for secure mediation. We present a Java implementation of a system that mediates SQL queries. Security risks for the client and the mobile code are delineated; offending the integrity of its own data is identified as a special type of attack of mobile code in a mediation system. We name appropriate countermeasures and describe the amount of trust needed in our system. As an extension, we consider security in a hierarchy of mediators. Finally, we combine mobile code with mobile agent technology.

1 Introduction

In a world with a growing amount of digitalized information, finding relevant data is a tedious task – and it gets even more difficult if the information is distributed on several different host systems (the "datasources"). Wiederhold and Genesereth (cf. [13,14]) introduced the concept of mediation to support the client of a distributed information system: The client directs a query to a so-called mediator; the mediator tries to gather the data best fitting the client's interests by sending partial queries to datasources; finally, the mediator constructs a global result out of the partial results and sends it back to the client. See Figure 1 for a basic mediated system.
This basic system does not consider security aspects. However, participants may require some security demands to be fulfilled:

1. Anonymity of participants: Clients may wish to stay anonymous to the datasources.
2. Confidentiality of data: Datasources may wish to be sure that the requesting client is eligible to access the requested data; i.e., datasources have to perform some kind of access control.

* This work was funded by the German Research Council (DFG) under grant number BI 311/11-1.

S. Jajodia and D. Wijesekera (Eds.): Data and Applications Security 2005, LNCS 3654, pp. 267–280, 2005.

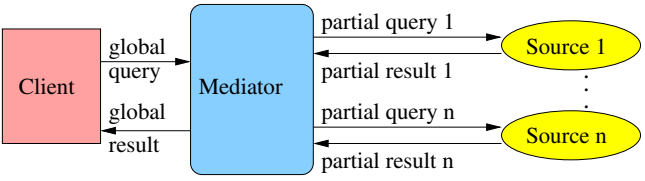

Fig. 1. A basic mediated information system

Altenschmidt et al. (cf. [2]) designed a system for secure mediation. Its general design is shown in Figure 2 and outlined in the following. Given that not revealing a client's identity is one precondition to ensure anonymity for the client (Point 1), in the secure mediated system a client attaches a credential to her global query; the mediator forwards the credential with the partial queries to the datasources. This credential is issued by a trusted certification authority (see Section 5 for a description of our trust model); it links properties of the client to her public encryption key but does not contain details of her identity. The client keeps another certificate linking her identity to her public key in a safe place. Instead of specifying just one key, the client can also attach a set of credentials (containing different properties linked with possibly different public keys of hers) and therefore combine multiple properties.

Datasources base their access control decisions only on the properties presented in the credentials. To keep the data confidential each datasource applies a hybrid encryption scheme to its partial result: A session key used to encrypt the partial result is itself encrypted with the client's public keys in the credentials. Assuming a reliable public key infrastructure and adequate encryption techniques, only the person who possesses the private decryption keys should be

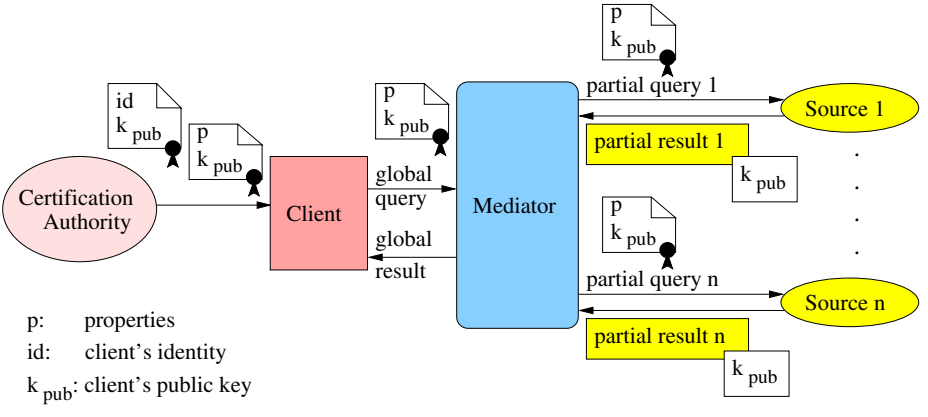

Fig. 2. A credential-based secure mediated information system

able to decrypt the session key and therefore the partial results and to access the returned data (i.e., Point 2 is fulfilled).

By encrypting the partial results we face the following problem that will be treated in this paper:

Given encrypted partial results, the mediator has to supply the client with a global result to her global query. However, the mediator may not be eligible to access the datasources' unencrypted data.

Unfortunately, there does not exist a general "privacy homomorphism" (as introduced in [11]) to solve the problem of "computing with encrypted data" (cf. [12]) where all data are encrypted with the same key. Furthermore, there are no approaches known to us that cover the problem of computing with data encrypted with different session keys; this is the case if the datasources apply a hybrid encryption scheme. So, up to now the mediator is not able to compute an encrypted global result from encrypted partial results. Moreover, the client should better not give away her private decryption keys as she cannot control what other parties use them for. Thus we have to impose the additional work of computing the global result on the client.

Section 2 introduces mobile data and mobile code as possible solutions to our problem and lists advantages of mobile code. Section 3 presents our Java implementation of the secure mediated system with mobile code. In Section 4 possible attacks by mobile code are depicted and a number of protection mechanisms are explained; attacks on the code are considered as well. Section 5 describes the model of trust we assume in our system. Section 6 covers additional security aspects in a hierarchy of mediators. We conclude the paper with an outline of another application area for a secure mediated system with mobile code. More details on our mediation system can be found in [16]; our Java implementation can be downloaded from [15].

2 Communication Between Mediator and Client

The basic idea to solve our problem is that the client gets the encrypted partial results from the mediator and additionally an instruction how to combine the partial results. We call the set of unencrypted partial results $\mathcal{R}$, the set of encrypted partial results $\mathcal{R}_{enc}$, and the combination instruction i. The client first decrypts the partial results and then computes the global result according to i; she uses a program for this computation. We distinguish two possible origins of this program:

1. A general program p was on the client's computer beforehand. The mediator specifies a format for i and $\mathcal{R}_{enc}$ and sends $d = (i, \mathcal{R}_{enc})$. On client side $\mathcal{R}_{enc}$ is decrypted to $\mathcal{R}$; then, i and $\mathcal{R}$ are used as inputs to compute the global result $p(i, \mathcal{R})$.
2. The mediator constructs a specialized program $p(i)$ based on i containing libraries that are necessary to compute the global result. The mediator then

sends $c = (p(i), \mathcal{R}_{enc})$. On client side $\mathcal{R}_{enc}$ is decrypted; the execution of $p(i)(\mathcal{R})$ yields the global result.

In general, the first approach corresponds to the concept of "mobile data", the second one to the concept of "mobile code"[1]. Similar to [1] and [10] we use the following definitions in this paper:

Definition 1. ***Mobile data, data processor:***
A mobile data system comprises a client requesting information and a server supplying this information. Communication between client and server is merely an exchange of data without program code. Processing the data on client side is solely done by a program already residing on the client's computer; this program is called a data processor. The server has to transmit the data in a predefined format.

Definition 2. ***Mobile code, execution environment:***
Mobile code consists of an executable program; there may also be a set of data included that the program operates on. An installation of the program is not needed; i.e., it can be executed immediately. A mobile code instance is sent to one single recipient only. To control mobile code, it should be run in an execution environment that shields the operating system from the mobile code. There is no need for a common data format. However, the mobile code has to provide a public execution interface to the recipient.

A pre-installed data processor on the client's computer is indispensable in the concept of *mobile data*. But also if we want to impose security settings on *mobile code*, a pre-installed execution environment is necessary. We opted for the mobile code approach as it offers some advantages that are described in the following.

In general, Fong [5] and Peine [10] consider mobile code advantageous over mobile data. Yet, their conclusions do not entirely apply to secure mediation. We now give just a short overview of our reasoning; see Table 1 for a summary and [16] for a detailed analysis.

In secure mediation, there is no difference between mobile code and mobile data considering **Distribution of state** and **Reliability of network**: Only one transmission from mediator to client is necessary, because the mediator collects all partial results; that is why inconsistencies in the distributed computation state or communication problems due to an unstable communication link hardly occur – be it with mobile data or mobile code. As for **Network traffic**, mobile code even increases the amount of transmitted data due to its additional libraries. However, mobile code offers the following advantages for secure mediation:

- **Locality:** Local interaction with resources (like the decryption keys for partial results) without further installation of libraries can only be guaranteed with mobile code.

[1] We also use the term "mobile code" in contrast to "mobile agents". The most distinctive feature of mobile agents is their ability to deliberately change their location in a network of host computers.

Table 1. Mobile code versus mobile data; gen. = general, med. = mediated, + = mobile code better than mobile data, 0 = equal, – = mobile code worse

Comparison	gen.	med.	Reason
Avoiding distribution of state	+	0	No inconsistency: Serial computation between mediator and client
Decoupling from network	+	0	Only one transmission: Mediator collects all partial results
Reduction of network traffic	+	–	Additional program binaries (e.g. class files)
Local interaction with resources	+	+	Computation at client without further installation
Footprint size	+	+	Small execution environment
Extensibility	+	+	Libraries are included in mobile code

- **Size:** The execution environment of mobile code may be considerably smaller than a data processor of mobile data as all necessary libraries are sent along with the code; those libraries occupy space on the client computer only while being used.
- **Extensibility:** The mobile code can use its up-to-date libraries without further need for action of the client.

3 Our Solution: Mobile Java Code

The Java implementation of our secure mediation system processes queries in the Structured Query Language (SQL). It comprises a client module (the execution environment for mobile code), a mediator module and a datasource module. The modules communicate via RMI calls. We describe each module in detail in the following subsections.

3.1 Datasource Module

The datasource module needs access to an appropriate Java database driver. When the datasource module receives a partial query, it opens a SQL connection to a database with the database driver to get the partial results. The datasource module encrypts a partial result with hybrid encryption. This has two advantages:

- Each partial result is encrypted with a newly generated symmetrical session key; this makes cyphertext-attacks aimed at recovering the client's private decryption keys more difficult.
- The possibly large set of data in the partial result is encrypted with the faster symmetrical encryption and only the small-sized session key is encrypted asymmetrically.

The default Java packages do not support asymmetrical encryption. However, it is possible to integrate so-called "cryptography providers". We used the "Bouncy Castle Provider" (BCP; see [4]). The session key is encrypted with the Bouncy Castle RSA algorithm (in Electronic Code Book mode and with Optimal Asymmetric Encryption Padding). Both asymmetrical encryption of the session key (with possibly a set of different public keys extracted from the client's credentials) and symmetrical encryption of the partial result are both carried out by the class `javax.crypto.Cipher`. The parameters for symmetrical encryption (algorithm and keylength) can be set in the GUI by a datasource administrator.

On client side, Java class definitions can be integrated at runtime. That is why datasources are not restricted in what data formats they use to represent their partial results. Their formats just have to implement the interface `Result` that is known to the client module; any format implementing `Result` can be processed on client side if the class definitions are sent within the mobile code.

3.2 Mediator Module

The mediator module uses the "SQL2Algebra" library developed at our department to process a client's SQL query. The library takes the query as input and outputs a so-called algebra tree. The leaves of this algebra tree contain the partial SQL queries that are forwarded to the datasources. Each inner node represents one of the algebraic operators *selection*, *projection*, *union*, *join* and *complement*; only queries representable with these operators can be processed by the library. This is an examplary SQL2Algebra transformation:

SQL query	Algebra tree

`Select distinct tv1.A`
`from TABLE1 tv1`
`union (select`
`distinct tv2.A`
`from TABLE2 tv2);`
	`UNION`
	`|_ PROJECT{A}`
	`| |_ "Select * from TABLE1;"`
	`|_ PROJECT{A}`
	`|_ "Select * from TABLE2;"`

Based on the algebra tree, the mediator module constructs the so-called "answer tree" – the executable that is returned to the client. Each inner node of the answer tree is an operator object; it provides a method that executes the respective operation on its child nodes. Similar to the data format of the datasources, the mediator can use operator classes unknown to the client as long as they implement the interface `ResultOperator` and their class definitions are included in the mobile code. A leaf of the answer tree is an object of type `ResultProxy`; it has a reference to a `java.util.Hashtable` that stores all encrypted partial results returned by the datasources. Then the mediator module constructs the mobile code by joining the answer tree and the necessary class definitions (as one or more Java Archives (JARs)) in an object of type `ResultExtractor`.

The mediator module encrypts the mobile code hybridly using the public keys contained in the client's credentials. Finally the mediator module signs the mobile code using `java.security.Signature`. Encrypting and signing is done to secure the mobile code during transmission between mediator and client.

3.3 Client Module

A client enters a SQL query and the mediator name (or its IP address) in the client module. She loads her credentials from a Java KeyStore (JKS); at the same time, the client module verifies whether she knows the password that secures the corresponding private decryption keys. The client module sends the query together with the credentials to the indicated mediator module.

After receiving the mediator's answer, the client module checks the signature of the mobile code with the verification key of the mediator that the client loaded from a JKS. If the signature is correct, the client module decrypts the mobile code with the private keys specified when loading the credentials. Decryption leaves the answer tree and the JARs. For each mobile code the client module creates a uniquely named working directory, where it temporarily stores the JARs. Then the client module accesses the `Hashtable` of the mobile code that contains the encrypted partial results, decrypts each partial result and writes the decrypted partial result back to the `Hashtable`.

Before execution is started, the JARs have to be made available to the Java classloading mechanism. We replaced the default system classloader with a new classloader that allows to add filenames to its search path and remove them again. As our classloader replaces the system classloader, simply the `new`-operator can be used in the mobile code to instantiate an object. So, the client module adds the JARs to the classloader's search path and calls the calculation method of the root operator of the answer tree. Each operator recursively calls the calculation methods of its children and then processes their return values. If a child is a leaf (i.e., a `ResultProxy`-object), the calculation method accesses the `Hashtable` containing the decrypted partial results and returns the appropriate one. The client module presents the global result to the client. After this, it resets the classloader's search path and deletes the working directory.

4 Security Issues

In general, in a dynamic mediated system with constantly changing, unidentified participants there is no basis for mutual trust between the participants. The client can protect her computer against outside attacks from other participants by adequate means (e.g., firewalls or authentication mechanisms). The crucial point is that the client has to let code enter her computer to benefit from the code mobility.

In our system, we still rely on certain trust relations (see Section 5); however, our design goal was to minimize the amount of trust the client has to put in the mobile code (specifically the program $p(i)$) she receives. To achieve this, we

use an execution environment that takes care of a secure execution of the code. This implies that the client has to check the small execution environment for correctness once before using it (or the client trusts the execution environment instead).

A feature that distinguishes a mediated system from other mobile code systems is that a mobile code instance does not have one unique producer. Instead, the program part of the mobile code is constructed by the mediator while data parts are supplied by different datasources. That leads to a consideration of the following principals:

- the mediator (as producer of program $p(i)$ and sender)
- the datasources (as suppliers of data $\mathcal{R}_{enc}$)
- the mobile code c
- the program $p(i)$ inside the mobile code
- the data $\mathcal{R}_{enc}$ inside the mobile code (or $\mathcal{R}$ after decryption)
- the client (and her computer)

In the following two subsections we explore in what ways mobile code and the mediator could attack the client and what the execution environment can do to protect the client. In the third subsection, mobile code is considered as the victim of attacks of the client.

4.1 Mobile Code Attacks the Client

The client wants security criteria for her computer to be met. The basic requirements are *confidentiality* (of the data and programs on the computer), *integrity* (of these data and programs) and *availability* (of hard- and software on the computer). That means mobile code should not be able to spy out or corrupt data and programs or monopolize resources on the client computer.

In addition to plain espionage, corruption or monopolization there are three special types of behaviour of mobile code that could potentially lead to one or more of these attacks. Mobile code could

- **conspire with other mobile codes on the client computer:** Single mobile codes may seem harmless; but if several different codes are allowed to communicate on the client computer, they could carry out an attack in combination. As an example, let c_1 be a mobile code that is allowed to read a decryption key (e.g. one to decrypt partial results) but may not use a network connection, and let c_2 be a mobile code that can open a network connection but cannot read any data. It is a case of espionage if now c_1 communicates the decryption key to c_2 and c_2 sends the key via the network connection to another computer.
  As a second example, consider two mobile codes that monopolize the processing unit by permanently alternating calling procedures of one another. This would be a denial of service attack on the availability of the client computer.

- **masquerade as another identity:** If a mobile code succeeds in convincing the client that it represents a trusted identity, it could misuse this trust for starting attacks. A mobile code could pretend to be sent by a trusted mediator although its real sender is an attacker unknown to the client. The client possibly would run such code with less restrictions.
  Mobile code could also masquerade as a part of another program. It could for example simulate belonging to the execution environment by opening a similar looking input dialog; in this dialog the mobile code could for instance ask the user to enter the password that secures a decryption key.
- **download other programs or program parts:** If a mobile code is allowed to receive data via a network connection, it could download additional programs or program parts that eventually attack the client. Young and Yung (cf. [18]) call this a "malware loader".

Literature on mobile code considers the following techniques to protect a client from those attacks (see e.g. [5,8]):

1. **Dynamic Access Control**
2. **Signed** or **Certified Programs** in combination with contracts
3. **Program Checking**; e.g. **Proof-Carrying Code** (see [9])
4. **Sandbox**

Which (or which combination) of these techniques is appropriate for a secure mediated system?

For the client, *proof-carrying code* would possibly be a good solution: She would have to check the proof for correctness before starting the code but would not have any performance loss due to dynamic checks. However, there is the difficulty that the mediator generates the code (e.g. the SQL algebra tree) at runtime and therefore also has to generate the proof at runtime. Since proof generation generally is more complex than proof validation, the client would have to wait quite a long time for the mediator's result.
Nevertheless there would be the following remedy if the code is constructed from modular building blocks – as for example the SQL algebra tree is constructed from only a small number of basic algebraic operators: The mediator could have the proofs for the building blocks ready and just prove that they are combined correctly for the particular query. Unfortunately, automated proof generation is still a field of intensive research; it is so far impossible to generate a proof for an arbitrary program. That is why we have chosen another strategy explained in the following.

As for the mobile Java code, some security-relevant operations have to be performed on client side:

- the secret decryption keys have to be accessed to decrypt the mobile code and the partial results
- a working directory for the mobile code has to be created and the JARs have to be saved there
- the filenames of the JARs have to be added to the system classloader's search path and removed again after execution

- the mobile code has to be started (i.e., system resources like the processing unit or memory have to be assigned to the code)
- the working directory has to be deleted after execution.

If the mobile code carried out all this operations, it probably would need a lot of changing permissions; i.e., *dynamic access control* had to be performed on client side. In our design however, we let the client module do all security-relevant operations and let the mobile code run in a *sandbox*. More precisely the mobile Java code does not need any Java permissions on the client computer; just the client module (as the execution environment) gets a minimal set of permissions to access the decryption keys, save the JARs etc.

The sandbox is a sufficient basis for execution of mobile code in our secure mediated system. However, a more advanced implementation could exploit advantages of dynamically associating different mobile codes to different Java protection domains (and by that assigning code different Java permissions). This would enable the client to give some codes a higher priority or let some codes communicate while others are not allowed to do this. Possibly execution of mobile code could also be based on time-dependent conditions. If the client performs some computations regularly, mobile code could be denied execution at that time to avoid denial of service attacks. Similarly, a context-based condition could take other running mobile codes into consideration and could prevent a conspiracy.

In the sandbox, code is not executed on contract basis, but rather technical protection mechanisms are employed. However, *signing the mobile code* by the mediator (i.e., the code producer) is additionally used to check integrity of the code after transmission to the client.

4.2 Mediator Attacks Client

The mediator takes a central position in a mediation system. It has access to the global query and all credentials in it and constructs the mobile code. It can abuse this positions to attack the *semantic correctness* of the result; i.e., it constructs a program $p(i)$ that computes a wrong global result.

We did not include a countermeasure for this attack in the implementation but we suggest the following adoption of the **proof-carrying code** technique to detect such an attack. While in the original application area (see [9]) the proof states that the code does not harm the client, now the proof attests that the mobile code contains a correct result to the client's query.
Take the SQL algebra tree as an example: The mediator could construct a forged algebra tree by exchanging algebraic operators (e.g. a *join* instead of a *union*). With code carrying a proof of correctness, the mediator has to prove that the algebra tree has been correctly derived from the query. This could be done by a proof that restores the query from the algebra tree.

Apart from that, a more subtle attack is possible: As mentioned before, in a secure mediation system the program $p(i)$ is produced by the mediator and the data $\mathcal{R}_{enc}$ are supplied by several datasources. On client side, $p(i)$ operates on the decrypted data $\mathcal{R}$. The program $p(i)$ could again attack the *semantic correctness*

of the global result and also the *integrity* of $\mathcal{R}$ while being executed on the client computer. In the Java implementation e.g., the operator classes inside the mobile code are unknown to the client. They process decrypted partial results $r \in \mathcal{R}$. The mediator could construct operators that compute an incorrect global result by changing values in every r.

The mediator could use traditional proof-carrying code or a certified program to assure the client of the correct execution of $p(i)$.

4.3 Client Attacks Mobile Code

The mobile code producer makes program $p(i)$ available to the client so that she can receive the global result; yet, the code producer may want $p(i)$ to be secured from infringement of *copyright*.

Several approaches have been made to address this problem. Software mechanisms are **obfuscating** (treated theoretically in [3]) and **computing with encrypted functions** (cf. [12]). Copyright protection in these forms is contradictory to program checking approaches used to protect the client from attacks of the mobile code (see Section 4.1); even proof-carrying code has to be processed by the client in clear to deduce a safety predicate. Especially Java byte code is difficult to protect due to the existence of decompilers and code purifiers. To overcome this problem, hardware components could be used in combination with certified programs (or certified proofs) to build a **trusted computing platform** (cf. [17]) but we have not considered this in our implementation.

5 The Trust Model

Our aim was to reduce the necessity for trust to a minimum. Yet, some trust requirements remain. The certification authority (CA) has to be trusted by all other participants (clients, mediators, and datasources) because all of them depend on its impeccable behaviour:

- A datasource has to trust that the CA issues correct and valid credentials such that only eligible clients are able to decrypt its data.
- Similar to the datasources, a mediator has to trust that the CA issues correct and valid credentials such that its mobile code is protected from access by clients other than the eligible one (or from access by the CA itself).
- A client has to trust the CA that it keeps her identity a secret.

Additionally, as long as copyright protection (as described in Section 4.3) is not put into practice, the mediator has to trust the client, that she does not use the mobile code in other ways than the mediator intended her to do (e.g., that the client follows a licence that accompanies the mobile code).

Equivalently, a datasource has to trust the client that she does not pass data on to other, non-eligible participants. However, a datasource does not need to trust a mediator; a mediator cannot attack the confidentiality of partial results if an appropriate encryption is used.

The main point is that the client has to trust the datasources (as suppliers of data) and the mediator (as producer of program code) just as far as *semantic correctness* of the global result is concerned because the execution environment protects her computer from other attacks (see Section 4.1). So, there remain two possibilities of attacks on the correctness of the global result: The data inside the mobile code might be semantically wrong (i.e., the client has to trust that the datasources supplied correct information; however, this is the same with any traditional database query), or the mediator might construct a mobile code that computes an incorrect global result (see Section 4.2; however, in basic ("unsecure") mediation the mediator could also corrupt the data supplied by the datasources).

6 Hierarchy of Mediators

As an extension to our system, we considered a hierarchy of mediators. The client still sends her query to one mediator, but mediators are able to forward partial queries to other mediators. This technique offers increased flexibility and scalability: Specialization of mediators to certain topics is possible; a mediator can also decide whether it forwards partial queries to more specialized mediators or just gathers partial results from its own datasources. In a hierarchy of mediators the mobile code is built by different code producers: Each mediator constructs a partial code containing its own program part, encrypted partial results and possibly other partial codes.

Since the execution environment protects the client from attacks (except the semantic ones) by mobile code of a single mediator, mobile code of a hierarchy of mediators does not mean an increased risk to the client. However, signature checking becomes complex when each partial code has been signed by a different producer; likewise a safety proof for proof-carrying code has to be combined from several partial proofs.

With a mobile code composed of different partial codes, not only attacks on the integrity of the data $\mathcal{R}$ are possible (see Section 4.2), similarly the *integrity of execution* of a partial code could be endangered by other partial codes. This problem is e.g. inherent to the Java classloading mechanism: In the JARs brought along with the mobile code, overwriting of class definitions can occur. A class is loaded from the JAR that is searched first; if a second JAR contains a class with the same class name, this second class definition is ignored. In our Java implementation, JARs are scanned for duplicate class definitions.

7 Conclusion

With the adoption of mobile code for secure mediation we are able to transmit data from datasources to a client in encrypted form; the mediator does not process any clear-text data. That ensures confidentiality of the data and reduces the necessity for trust in the mediator. Our mobile code system is easily extensible and its execution environment is small.

Optimizing runtime performance was not in the main focus of this work. In comparison to basic ("unsecure") mediation, in our secure mediated system performance penalties occur mainly due to encryption on datasource side, mobile code generation (and again encryption) on mediator side and decryption and mobile code execution on client side. Due to a lack of time, runtime performance has not been investigated systematically. Nevertheless, test runs performed in an acceptable amount of time. One possibility of reducing both encryption time and execution time would be to improve the SQL2Algebra-library that generates the algebra trees on mediator side; algorithms that minimize the size of partial results and the depth of the tree could be included.

Our mediation system with mobile code could be combined with existing mobile agent systems (see e.g. [10,7]) to profit from both technologies: The client directs her query to a mediator; the mediator constructs an agent to collect the partial results from the datasources. This reduces communication overhead between mediator and datasources. Wrapping functions could be carried out by such an agent as well to convert partial results into a homogeneous format. The client, however, is not charged with agent creation, agent management etc., since the mediator takes care of all agent-related actions. Security considerations for mobile agent systems have been investigated profoundly (see e.g. [1,6]).

References

1. Joy Algesheimer, Christian Cachin, Jan Camenisch, and Günther Karjoth. Cryptographic security for mobile code. In *SP '01: Proceedings of the IEEE Symposium on Security and Privacy 2001*, pages 2–11. IEEE Computer Society, 2001.
2. Christian Altenschmidt, Joachim Biskup, Ulrich Flegel, and Yücel Karabulut. Secure mediation: Requirements, design and architecture. *Journal of Computer Security*, 11(3):365–398, June 2003.
3. Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. *On the (Im)possibility of Obfuscating Programs*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–19. Springer, Berlin, 2001.
4. The Legion of the Bouncy Castle. `http://www.bouncycastle.org/`.
5. Philip W.L. Fong. *Proof Linking: A Modular Verification Architecture for Mobile Code Systems*. Phd thesis, Simon Fraser University, Burnaby, Canada, January 2004. See `http://www.cs.sfu.ca/research/publications/theses/`.
6. Günther Karjoth, Nadarajah Asokan, and Ceki Gülcü. *Protecting the Computation Results of Free-Roaming Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 195–207. Springer, Berlin, 1998.
7. Günther Karjoth, Danny B. Lange, and Mitsuru Oshima. *A Security Model for Aglets*, volume 1419 of *Lecture Notes in Computer Science*, pages 188–205. Springer, Berlin, 1998.
8. Sergio Loureiro, Refik Molva, and Yves Roudier. Mobile code security. In *Proceedings of ISYPAR'2000 (4ème Ecole d'Informatique des Systèmes Parallèles et Répartis)*, pages 95–103, Toulouse, France, 2000.
9. George C. Necula and Peter Lee. *Safe, Untrusted Agents Using Proof-Carrying Code*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer, Berlin, 1998.

10. Holger Peine. *Run-Time Support for Mobile Code*. Dissertation, Universität Kaiserslautern, Fachbereich Informatik, October 2002.
11. Ron L. Rivest, Leonard Adleman, and Michael L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, pages 169–179, 1978.
12. Tomas Sander and Christian F. Tschudin. *Protecting Mobile Agents Against Malicious Hosts*, volume 1419 of *Lecture Notes in Computer Science*, pages 44–60. Springer, Berlin, 1998.
13. Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.
14. Gio Wiederhold and Michael Genesereth. The conceptual basis for mediation services. *IEEE Expert Intelligent Systems and their Applications*, 12(5):38–47, September/October 1997.
15. Lena Wiese. Mediator with mobile code support. `http://ls6-www.cs.uni-dortmund.de/issi/projects/DFG_Kompositionalitaet/mobilecode.html.en`.
16. Lena Wiese. *Sichere Mediation mit mobilem Code – Implementierung und Sicherheitsanalyse*. Diploma thesis (in German), Universität Dortmund, Dortmund, Germany, October 2004. See `http://ls6-www.cs.uni-dortmund.de/issi/archive/literature/2004/Wiese_2004.pdf`.
17. Bennet Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. In *Proceedings of the First USENIX Workshop of Electronic Commerce*, pages 155–170, Berkeley, CA, USA, 1995. USENIX Assoc.
18. Adam Young and Moti Yung. *Malicious Cryptography – Exposing Cryptovirology*. Wiley, Indianapolis, Ind., 2004.