Guarding Security Sensitive Content using Confined Mobile Agents

Guido van 't Noordende
Vrije Universiteit
De Boelelaan 1081
1081HV Amsterdam
The Netherlands
guido@cs.vu.nl

Frances M.T. Brazier
Vrije Universiteit
De Boelelaan 1081
1081HV Amsterdam
The Netherlands
frances@cs.vu.nl

Andrew S. Tanenbaum
Vrije Universiteit
De Boelelaan 1081
1081HV Amsterdam
The Netherlands
ast@cs.vu.nl

ABSTRACT

Mobile code and mobile agents are generally associated with security vulnerabilities, rather than with increased security. This paper describes an approach in which mobile agents are *confined,* in order to allow content providers to retain control over how their data is exported while allowing agents to search the full content of this data locally. This approach offers increased control and security compared to the traditional client-server technologies commonly used for building distributed systems. We describe a new system, called Mansion, which implements confinement of mobile agents, and describe a number of applications of the confinement model to illustrate its potential.

Keywords

Mobile Agents, Confinement, Information Flow Control

1. INTRODUCTION

Current distributed system architectures such as the web, web services, and distributed object systems [1, 2], are essentially client-server based. These systems often provide some security mechanisms such as client or user authentication [3, 4], but are weak at enforcing information flow control policies. As soon as a client has access to a document or data file, its contents are vulnerable in that the client can redistribute this information to unauthorized parties. A number of approaches attempt to minimize the risk of such unwarranted dissemination of security sensitive content. Digital Rights Management (DRM) [6] is an approach which can prevent unwarranted content redistribution by using secure hardware on the client side. However, such hardware may not always be available, must be authenticated, and may not be fully trusted by the content provider. Therefore, DRM may not be feasible in all applications. Some detection mechanisms (such as watermarking [5]) are being developed which do not depend on secure hardware, but instead allow for detection of information leakage once it has occurred by including information in the data which can be used to detect leakage after the fact, and/or provide information leading back to the principal that leaked this information. However, for certain types of data, it may be impossible to include such information in the data without compromizing usability of this data. In addition, detection of leakage after the fact may not be enough for some applications. For example, medical data may contain privacy sensitive information that should never be exposed to unauthorized third parties. As a result, security sensitive data with dissemination constraints is most often not remotely accessible. For example, medical data is generally stored in off-line medical information sytems, and only accessible to authorized hospital staff members in the hospital building. Similar constraints apply to other confidential or security sensitive information.

However, the closed nature of information systems containing security sensitive information poses restrictions on its use: for example, making particular hospital data (partially) accessible in electronic format could allow researchers (e.g., doctors or biologists) to obtain valuable information on the occurrence of certain diseases in particular geographical areas, or would allow, for example, (retrospective) epidemiological studies on much larger data sets than is currently possible. Other examples of information for which dissemination control is useful are music, pictures or movies, and other intellectual property in electronic format.

Mobile agents are mobile software programs which can be programmed by, and act *autonomously* on behalf of, their owner to achieve some goal [7]. In contrast to client-server programs, mobile agents can migrate to the location where the data resides, and search the available data locally there. Often cited benefits of using mobile agents are decreased network usage, as data does not have to be transported over the network before it can be used by a program (agent); decoupling agent execution from home nodes[1]; and spreading the computational nodes over multiple machines - in particular when submitting multiple agents in parallel. Another reason for using mobile agents instead of client-server technology, is that data owners can keep control over their information while it is being searched by a mobile agent on their own machine; only when selected data is being exported (e.g., through a communications channel) out of the content provider's control, does data filtering, dissemination protection or payment issues come into play. Finally, mobile agents offer *flexibility* in the way in which content is being searched, as compared to client-server technologies.

Often, client-server architectures for content retrieval use a query interface at the server side, which provides a special query language (e.g., SQL) in order to find data matching certain criteria. However, query languages are not very flexible with regard to matching criteria and with regard to the data type (e.g., text) for which they are designed, and may be unsuitable or overly specialized in certain application domains such as for matching DNA sequence data [8]. This

[1]E.g., a user can launch an agent into the network prior to boarding a plane, shut down the laptop, and collect the agent with its results after landing.

SAC'07 March 11-15, 2007, Seoul, Korea

means that, in certain applications, it may be hard to select data using a query interface in a precise enough manner. Correct matching is of particular importance in situations in which cost and/or confidentiality risks play a significant role. For example, a program that attempts to calculate the evolutionary distance between a template DNA sequence and a sequence stored in a remote data base, requires the full DNA sequence from the data base to make its comparison. At most, a query interface or index can be used to pre-select DNA sequence files, but after that the file has to be shipped to the client for comparison. At that time, it leaves control of the data owner, and either the data needs to be purchased, or it has to be marked in some way to avoid unwarranted dissemination by the client (note that watermarking is not likely to be applicable to DNA sequence data) in the case where the data file is considered propriatory. However, of this pre-selected data, the client may actually need only a subset that matches its query (e.g., the set of sequences which lie within a bounded evolutionary distance from the template DNA sequence), as a result of which the client obtains too many data items.

Mobile agents provide an alternative. Mobile agents can search through (potentially large) data sets locally, avoiding the transfer of large data sets over the network and allowing fully customized search on raw data. In this case, the client purchases (or signs a non disclosure contract for) only the data which actually meets its requirements, based on fully client-customized search on, for example, DNA sequences. Other examples of applications in which customized search is useful are described in Section 4.

This paper proposes a model, *mobile agent confinement*, which allows (authorized) users to search for information in confidential or privacy-sensitive remote data collections, without obtaining a local copy of this data. After selecting a set of data items, application and data specific filtering algorithms implemented by the data owner can be applied to the data, to determine which, and in what format and under which conditions, data can be exported from the data owner's machine to the agent's owner.

The idea of using mobile agent confinement to protect intellectual property is not new. In [8], Belmon and Yee describe a model where agents can search data locally on machines owned by the intellectual property holder, and where agent owners pay for the data they transmit back home. However, their paper is conceptual and provides no concrete implementation details; in addition, their proposal does not cater for data filtering at the server side, and does not prevent covert channels. Instead, the authors focus on an economical model where the agent pays a charge per bit (or byte) depending on the value of the data which has thusfar been inspected by the agent, and covert channels are treated only by charging for their potential use. This model, therefore, cannot be used for applications in which information leakage must be completely prevented. In contrast, this paper proposes a general framework, usable by multiple applications, where agents can be confined prior to accessing sensitive data on a server machine, and application (and data) specific filters can be applied to the data before it is transmitted back to an agent's owner. An implementation for secure confinement of mobile agents which prevents data leakage through covert channels is presented in Section 2.

Most mobile agent research so far has focussed on agent (system) programming, rather than on how to provide generally applicable, scalable infrastructures that support applications which use mobile agents. In particular, there does not exist a clear model or widely deployed infrastructure in which mobile agents can be launched to find information, and to which information can be added in a straightforward way. Much research has been done on protecting infrastructure and machines against possibly hostile mobile agents [9], but few agent systems to date have provided generally applicable application-level security mechanisms. The Mansion paradigm, and its confinement mechanism as explained in this paper, fills this gap.

Agent confinement is designed as an integral part of the Mansion system [10], which aims to provide a clear paradigm for designing distributed, secure mobile agent applications. Content can be placed in the Mansion system in so-called *rooms*[2] without requiring any central control; if this content is security sensitive, it can be placed in a room specially marked (by its creator) as containing sensitive content. In such a room, agents are automatically confined by the Mansion middleware to avoid export of information from this room except through a single, data owner provided, data export mechanism. The Mansion paradigm, its middleware, and the implementation of its confinement mechanism is described in section 2.

For the confinement model to work, agent mobility is a prerequisite: the security provided by confinement cannot generally be obtained using traditional client-server technologies. The Mansion middleware system contains the necessary mechanisms to enforce agent migration and confinement securely. Agent confinement is part of the conceptual model of Mansion, and can be used in any application built using Mansion.

This paper describes the implementation of mobile agent confinement in Mansion. Using confiment, agents can inspect remote data collections in a controlled way, such that agents can only take information with them (to return to their owner) when this is allowed by a policy defined by the owner of the data collection. Section 2 describes the Mansion framework and its implementation. Section 3 introduces the notion of confined rooms. Section 4 illustrates the potential of the framework in a number of different application domains. Section 5 compares our approach to others reported in the literature. Section 6 discusses the results and identifies areas for future research.

2. MANSION PROGRAMMING PARADIGM AND ARCHITECTURE

Mansion [10] is a mobile agent system which allows agents to roam through a world in which data is stored in objects. Mansion offers an application programming paradigm based on the notion of *hyperlinked rooms*, which contain the abovementioned objects and which can be visited by agents so that they can search through the content of the room locally. Each room contains hyperlinks to other rooms, which agents must use to migrate to another room. All objects and hyperlinks are annotated using sets of attribute-value strings, so that agents can find their way in the system in order to find information that may be interesting to them. In addition, agents can see each other in a room to meet each other there, so they can communicate with each other to exchange information in order to speed up their search for suitable rooms and content, or to do business with each other directly.

Applications can be designed with specific properties. Different applications run as separate worlds, where each world contains a (potentially large) set of hyperlinked rooms. The world owner can define rules on how the rooms in a world may be hyperlinked in order to impose structure on the world. For example, a world may consist of a single *world entry room* which contains hyper-

[2]Superficially, a room is comparable to a web site, in that rooms can be created dynamically and without requiring central control, and in that content can be placed in those rooms without central control as well. However, the web does not support mobile agents, and Mansion's internal design is completely different from the web.

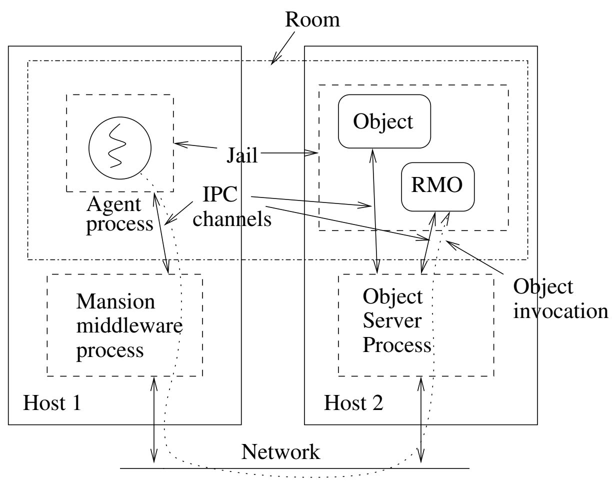

Figure 1: The Mansion architecture. An agent and middleware components required for running Mansion are shown. All middleware components (including agents and objects) are implemented as separate processes (dashed boxes). Agents and objects are executed as separate jailed processes (see text) to avoid interference or information leakage. A room (dash-dotted box) is composed of a Room Monitor Object (RMO) and the objects in that room. Agents invoke Mansion API methods and objects using RPC calls to the Mansion middleware process, which acts as a reference monitor for all invocations made by an agent and ensures that agents cannot access any objects outside their current room. A room can be physically distributed over multiple processes under single administrative control, possibly running on multiple machines. The path that a single invocation on an object (here, the RMO) follows is shown (dotted arrow); the marshalled reply of the object invocation takes the same route back.

links to all rooms in the world (forming a a tree like structure), or all rooms in a world may be allowed to contain hyperlinks to any other room in the world. However, the rooms in a world can be deployed autonomously by different administrative authorities, which each control their own sets of rooms and retain full control over the content of the (objects in their) rooms.

Objects can be one of a set of generic object provided as part of the Mansion system (e.g., a File Container object used for storing files), but an application (world) designer can add additional application-specific objects to the system. A room is implemented as an object (the Room Monitor Object (RMO)), which essentially functions as a registry for all the content of the room; agents are automatically connected to their current room's RMO (and only their current room's RMO) when they are started up by the middleware; agents invoke this object when they request information (e.g., query attribute-value sets) about the content of the room. Interagent communication also takes place through the Mansion middleware process. The middleware process may connect to a middleware process that may be in a different administrative domain, depending on where the peer agent runs, in order to properly handle a communication request.

Mansion comes with a specific Application Programming Interface (API). Using this API, agents can interact with their world, for example to communicate with other agents or to invoke objects, but they cannot communicate directly with programs outside the world. Mansion is language neutral: it provides a middleware program which runs on every machine in the world, which acts as a reference monitor, resource manager, communications manager and object broker for agents. The Mansion architecture is shown in fig. 1; details are elaborated upon in the remainder of this section.

2.1 Jail-Based Protection

For security reasons, we run all agents and objects in a protection system called a jail [11]. All system calls of a jailed process (e.g., an agent) are *intercepted* by a jailer process, which enforces a policy that confines the jailed process and prevents the local system from being harmed. The jailer implements the mechanisms needed to avoid that an agent can, for example, access the user's files or set up a connection to a process outside the Mansion system. The `ptrace` system call, available on almost all UNIX systems, provides the basic functionality needed to implement the jailer. By using ptrace, we avoid that the system requires changing the underlying operating system (which would hinder widespread deployment), and achieve portability to some degree. Earlier jailing systems that used ptrace were vulnerable to certain race conditions which would allow processes executing in them to bypass the jailer's security policy [12]. Our jailing system extends on this earlier work by using a novel approach to protect the system against these race conditions. Further details on the jailer's architecture and implementation can be found in [11].

An important reason for using jailing is that it is language neutral, as it works at the system call interface; even for interpreted languages like Java, Perl and Python, system calls are the eventual outcome of any operation which interacts with the outside world (e.g., file or network access operations). In addition, this approach makes it possible to use binary agents and objects compiled from C or C++ code in our system. In addition, an interpreted agent that makes use of native libraries (which could be a means for the agent to escape its language-based protection mechanism) can run under the same security constraints as other agents, by running them (together with their interpreter) as a process inside a jail.

Resource control mechanisms were also added to our jailing system to avoid an agent consuming a host's resources (such as CPU time or memory). By controlling resource usage and by avoiding that an agent can read global system state (e.g., execution times or memory usage of other agent processes running on the same machine), we have a handle to avoid most if not all covert channels which can be used by a malicious agent to export information to another process on the same or another machine. By disallowing agents to use resources above a certain treshold, and by disallowing agents to make system calls from which they can derive other agents' resource usage, it becomes very hard if not impossible to transport information from one agent to another, even when these agents are running on the same machine. To eliminate covert channels, all agent processes must be jailed, as is the case in Mansion[3].

Preventing covert (as well as direct) channels between agents is important when building a system intended to confine processes in order to protect against unwarranted information dissemination. For this use, all output channels should be prevented, not just the most obvious ones such as exporting information through shared files, TCP connections or interprocess communication[4]. The Mansion jailing system was designed specifically to avoid information

[3] An unjailed process can, for example, easily find out information about resource usage of a jailed agent by inspecting the /proc device in Linux. Accessing /proc is forbidden by the jailer's default security policy, except for a few safe, necessary subdirectories of /proc.

[4] Even direct communication channels are not considered well by most existing jailing systems, which are generally not designed for preventing information flow between jailed processes. For example, most jailing systems allow the use of certain IPC related system

leakage between jails. Our jailing system is, to our knowledge, the first to attempt a complete coverage of covert channels (as well as direct channels) which prevents information leakage between processes that run in different jails.

We believe that our jail design goes a long way to provide a practical, sound solution against information leakage through covert channels. The jailer has been implemented as a user-mode program on Linux (no changes to standard Linux are required to run the system). We have tested several nontrivial stand-alone and agent programs, including Java programs executing in a JVM, which could run without any problems inside a jail. Execution overhead lies between 10% and 200%, the latter for a program (*ant*, a Java build environment) which makes an excessive number of file system related system calls. The overhead imposed by jailing depends on the number and type of system calls made by the jailed program. Differences in overhead for different system calls are caused by the way in which the system calls (and their arguments) have to be processed by the jailer program, and are primarily caused by the mechanism used to prevent race conditions that could otherwise allow processes to bypass the jailer's security mechanisms [12, 11]. Based on an evaluation of performance data, most agent programs are expected to perform with an overhead between 10% and 50%, which we consider acceptable in relation to the flexibility and security achieved.

2.2 Mansion API Implementation

A fully closed confinement system is only usable if supplemented by an API which *may* be used by agents, which allows the agent do useful things in a controlled way. Mansion comes with an API which allows agents to do useful things in a Mansion world. Agents can invoke methods on the Mansion API by making RPC calls over a dedicated socket connection set up between their jail and the Mansion middleware program at agent startup time[5]. Agents are started up as jailed processes by the Mansion middleware process, and their life cycle (e.g., killing and suspending the agent) is also managed by this middleware process.

Agents can only interact with the outside world using Mansion API calls. The Mansion middleware implements the API, and acts as a *reference monitor* with regard to the agent's invocations on this API. An agent cannot connect to the RMO or to objects in another room than the one in which it currently resides. Agents cannot directly set up (TCP or UDP) connections to processes outside the system; the jailing system's policy denies the system calls necessary to set up such connections. Furthermore, the jailing system pre-allocates a private directory on the local file system for read-write access by the agent, which is not shared with any other process and which is cleared when the agent migrates to another room; this way, export of information through the local file system is prevented. Agents can set up connections to other agents through a Mansion API method, depending on authorization checks made by the middleware. Several other API methods exist, which are outside the scope of this paper; however, all methods are appropriately checked at the time of invocation by the Mansion middleware, and may be denied by it when appropriate.

Objects run as processes which are managed by a trusted object server process (see fig. 1), which may run on a different machine than the Mansion middleware. Each object process contains a generated skeleton interface and an implementation written in C++. Objects can only be invoked by agents via the Mansion middleware. To be able to invoke methods on an object, an agent must first *bind* to the object using a Mansion API method. Binding connects an agent to an object so that it can be invoked using a stub in the agent's address space. The Mansion middleware transparently forwards invocations to the object server where the object resides. Agents are compiled with a (generated) stub for each object type they may access in a world. Currently, only C and C++ stubs have been implemented, but it is straightforward to generate stubs for a different language, e.g., Java. Object interfaces are application (world) specific and stubs are generated from a language-independent IDL and provided as part of an agent programming library provided by the world designer. Agents invoke object methods using RPC calls over the same connection to the middleware that is also used for invoking Mansion API calls. Agents and objects are jailed for protection reasons (see fig. 1). Although object implementations are generally trusted, jailing avoids vulnerabilities in the object's code from exposing the local machine (e.g., file system) to attacks by an agent.

2.3 Secure Agent Migration

Agents are shipped into the Mansion system using an Agent Container. An AC is a simple, cryptographically protected, per-agent migratable file system for storing the agent's code and data. The AC's *segments* (files) can be either persistent (i.e., immutable and not removable) or transient (mutable and removable). Before migrating an agent, its AC is signed, and a secure audit trail is established by incrementally sigining all changes to the AC, each time it migrates [10]. The agent's owner signs the first signature of the AC's content (including its code and initial data segments) before it is shipped into the world. This signature functions for authenticating the agent when it is received by a Mansion middleware process, before starting it up. A secure handoff protocol (over a mutually authenticated SSL channel) exists for protecting the agent's audit trail against tampering.

Each object has an Access Control List (ACL), which determines which methods a particular agent may invoke based on the agent's authentication. A default entry may exist for unknown agents. The RMO's ACL determines if an agent may enter a room or not.

An important property of agent migration in Mansion is that agents are restarted completely when they follow a hyperlink, even if the agent can access the target room from the machine where it was already running. To retain knowledge of what its computational state is, an agent must write its found data and important parts of its internal state to its AC prior to following a hyperlink. Utility functions are provided by Mansion which can be used to write regular files to the AC. This simplifies 'self-serialization' in that, for example, an agent can use memory-mapped files to store process state prior to migration, which then can be straightforwardly copied to the agent's AC and recovered and remapped (by the agent) after migration. When an agent follows a hyperlink, the agent's process is killed and its AC is signed and sent over to a middleware process where the target room is accessible from. After the agent's AC is verified and authorized, and after it is restarted by that middleware, it can read its data segments and recover how far it got in achieving its task. This type of migration is called *weak migration*, in contrast to strong migration, in which the system takes care of reinstantiating the agent at exactly the same point in its thread of execution as

calls which make use of user defined tokens for access control. If agents pre-agree on such tokens, they can easily exchange information via such IPC channels, even if they are executed in completely unrelated jails. Our jailer does not prevent IPC channel usage, but verifies that an IPC token is not already in use and prevents agents in different jails to set up IPC channels to each other using pre-agreed IPC tokens.

[5]The TCP port that the agent may set up a connection to is specified as a commandline argument when starting up the agent, and the agent's jailer is configured at startup time to allow the agent to set up a connection to this port.

it was before migration [13].

Several issues make it hard to implement strong migration without specialized language or runtime support, in particular when moving agents to a different type of machine. Although solutions for strong code mobility were proposed in operating systems [14], in modified (Java) virtual machines [15], and by using a code translation, preprocessing or (byte)code rewriting approach [16, 17, 18, 19], none of these have reached significant deployment. All of these approaches also suffer from one significant drawback: even when strong mobility is implemented for a specific language or operating system, such a solution is inherently not portable to other languages or operating systems. Instead, by choosing weak migration, Mansion allows for agents written in different (interpreted) languages, and agents compiled for different platforms, to be shipped as alternative implementations within a single agent container, such that a platform that receives an agent can select an implementation which is appropriate for this platform[6]. Thus, even binary agents based on, for example, legacy code written in C or Fortran can be used, even when using Mansion in a heterogenous environment.

However, the most important reason for using weak migration in Mansion is that, this way, an agent forgets everything it did not write to its agent container when it follows a hyperlink: as nothing of an agent's execution state remains after migration, an agent cannot export any information by means of keeping it in memory; the only way to export information from a room is by sending it over a communication channel to another agent or by writing the information to its AC. This is crucial to the implementation of the confinement model, for reasons explained later in this paper.

3. CONFINED ROOMS

A *confined room* is a regular room in all respects, except that agents in a confined room cannot communicate with the world outside the confined room. In an *unconfined* room, agents can set up connections to other agents anywhere in the Mansion world, but this is not possible in a confined room. In addition, agents in a confined room cannot change, or write any data to, their agent container. As a consequence, an agent cannot export any information from a confined room. Within a confined room, agents can inspect all content of the objects in that room, and they can communicate with any other agent in that room. The idea is that agents thus can inspect, and search in full, any security sensitive content in a confined room. As agents (and the room and its objects) run under control of a Mansion middleware system which is deployed and trusted by the content owner, it is safe to place any content in the room.

Confinement is a property of a room, not of a Mansion application (world) as a whole: any room can be marked as a confined room at the time of its creation; other rooms in the world are not influenced by this room's confinement. Thus, the confined room is a security mechanism which is usable in any application by any room owner. In order to achieve confinement, the creator of a new room can simply mark it as a confined room when it creates the object that implements the room (see fig. 1). The middleware which

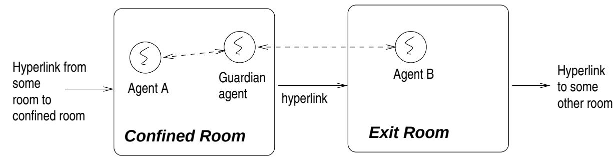

Figure 2: An example of a confined room in Mansion. Agent A is in a confined room and communicates its findings with the Guardian Agent (GA). Findings can come from objects or other agents in the confined room (not shown). Agent B is in an exit room and asks the GA for its findings from the confined room.

hosts the agent that is in this room[7], checks this property, and if the confinement property is set, it confines the agent at the time that it enters the room: the agent can now only access and exchange information within the room, but not with the outside world. When the agent leaves the confined room, it is automatically started up in a special *exit room*, without any recollection of what it did or saw in the confined room (see fig. 2).

To export information (possibly filtered by the content owner) from a confined room, a special *Guardian Agent (GA)* is placed in the room by the content owner. The GA is marked as such using a special attribute-value pair (see section 2) so that it can be found by agents in the room. Only the GA is allowed to communicate with agents outside the room, and it acts as a gateway to the world outside. Agents can export information by providing this information to the GA; after leaving the confined room, the agent can contact the GA from the exit room to obtain the information it provided (after GA specific filtering) from the GA. This is shown in fig. 2.

As the agent is now unconfined, it can store the information in its AC and transport it back to its owner. Data in the AC can be encrypted with the agent owner's public key to protect it from being readable during agent transport or on the remainder of the agent's itinerary. This GA can be programmed by the content owner to do any kind of filtering on the data or indices provided by an agent to it. As it is an agent, its filtering algorithm is highly customizable to match the content in the room, without requiring any changes to the Mansion middleware system[8].

For example, a confined room may contain patient records, stored in a file container object. Files have locally unique names, and when an agent finds an interesting record (e.g., the patient has a combination of symptoms possibly indicating a disease that a researcher is interested in), it can pass the file name to the GA[9] which can replace the patient's name or (personal) identifier(s) with a pseudonym, or blank out certain information. A new (random) identifier can be associated with each datum, as an identifier of the datum for later use which can only be associated with the real datum by the data owner. From the exit room, the agent can obtain the (now anonymized) data files so that its owner can inspect the information. In addition, a phone number of the treating physician may be provided to the agent by the GA, so that the researcher can contact the physician to ask for more information or permission to use the information, e.g., for setting up a trial or for a publication.

[6]Note that this allows for execution of Java agents under an appropriate Java security policy on those machines where a jailing implementation does not exist. However, note that the standard Java security model does not directly or automatically provide protection against covert channels, and cannot prevent information leakage from occurring in many cases, e.g., when native libraries are being used. Also, resource protection mechanisms are missing from current JVMs [20]. Therefore, this approach for executing Java agents in the absence of jailing should not be used for executing agents in a confined room.

[7]An agent can only be in one room at a time in Mansion, and can only access objects within this room

[8]The Mansion middleware has been implemented by us and can be used for any world; only agents and objects may have to be adapted to the requirements of a specific Mansion application.

[9]It is better to pass a reference to the GA than the actual file, as an agent may hide information in a data file, which is not possible when a filename is passed to the GA. A file container object generally provides read-only access to files.

Note that in the exit room, the agent has no direct access to the data in the confined room, nor does it have any recollection of what it did in the confined room; therefore, it is impossible for an agent to obtain any information from from the confined room except by asking the GA for this information.

4. USAGE EXAMPLES

Above we exemplified the use of a confined room with patient records. This section describes some additional examples of the use of confined rooms. We are currently working on a world with biological data, which uses the existing Mansion middleware infrastructure, in which we created some confined rooms containing propriatory biological (sequence) data owned by a fictional company. Obtaining DNA or RNA sequences or protein structure information is an expensive operation, and such information may be considered propriatory or may be only available for sale. However, before some researcher considers obtaining sequence data for a fee or under a non disclosure contract, it has to be determined if this data matches the researcher's criteria. Many sequence matching algorithms exists, many of those experimental in nature. The implementation of this application allows agents, with customizable algorithms for sequence comparison, to search for DNA sequences matching a template DNA sequence in a confined room. Potentially interesting sequence files are passed (by name) to the GA, and obtained from the GA when the agent leaves the confined room. The GA then passes URLs[10] to the agent for each matching file; this URL can then be used by the agent's owner to fetch the data file, either after payment or using a password obtained from an earlier (off-line) registration procedure, after the agent returned.

An application which has been described before is for multimedia databases or for image retrieval, where agents can search through images remotely to find images matching some sample image or thumbnail [21, 8]. Here, most emphasis goes to gains in efficiency due to decreasing network load and spreading computational load over the machines where the data resides [21], although security advantages have also been described. However, the existing literature does not provide a clear model with which content providers can place arbitrary content on network nodes and mark this content as being security sensitive such that selective confinement can be applied. More than that, existing models usually do not distinguish between logical containers of content (like rooms) and the physical infrastructure (network nodes, machines); because most mobile agent systems do not provide a clear separation between the logical level and the physical level, they often do not provide a clear view to the programmers and users on how and where to place content, and how agents can find this content. Also, the means to export information (e.g., using watermarking or through a payment scheme) is often hardwired to the described applications and systems, so that the described solutions lack generality. Instead, using the Guardian Agent combined with the logical model of confined rooms in a Mansion world, our confinement model allows for using the Mansion infrastructure to use confinement for any application, and to use completely customizeable data filtering or export-time data adaptation (e.g., watermarking) or payment schemes as required by the content owner.

Note that, in itself, confined rooms do not solve the problem of content redistribution; it merely allows for application-specific filtering of content prior to obtaining it, and for applying provider specified policies to data before it leaves the confined room. At that time, payment schemes, watermarking, or legal contracts still have to be applied. Such contracts or watermarking are essentially orthogonal to our solution and can (and generally should) be applied in addition to any confinement scheme to improve end-system security. However, as data selection by mobile agents running under confinement is much more fine-grained and much more customizable to a client's needs, we expect that the willingness for both end-users (clients) and data providers to agree on targeted redistribution contracts or payment for the found data is likely to be much larger than in systems where the client has incomplete control over selecting information, as this implies that larger and less suitable data sets are returned. Also, the risk of unwarranted dissemination of content by a client is much less when only a few items are exported than when a larger and less specific subset of sensitive or expensive data is exported. Because of this reduced risk factor, a larger amount of security sensitive content or intellectual property may become available to the general public or an authorized subset of the public[11], where this content would not be likely to become fully accessible using traditional client-server technologies[12]

5. RELATED WORK

A number of mobile agent systems have been described in the literature. Telescript [22] was the first commercial mobile agent system, and pioneered most of the concepts common to mobile agent systems. With the advent of Java at the end of the 1990s, a large number of Java-based mobile agent systems were built [7]. These systems are largely dependent on the security and platform independence provided by Java (e.g., [23, 24, 25]). Only a few systems support heterogenous agents. Notable examples are D'Agents [26], Ara [27], TACOMA [28], and AgentScape [29]. No mobile agent system to date uses jailing to protect the system against mobile agents.

In the existing literature, not many systems exist that use mobile agents, or mobile code, for confinement in a similar way to what was described in this paper. Belmon and Yee [8] are closest in nature, but they focus on an economic model for billing users for obtained data in commercial applications, rather than on application-specific filtering of data based on privacy or other security constraints. Roth, Pinsdorf and Peters [21] also touch upon security advantages of using mobile agents in an application implemented in the SeMoA framework [30]. However, the system described in their paper is limited to searching digital images using mobile agents.

Several system call interception based jailing systems exist [31, 32, 33, 34, 35, 36]. Some of these depend on modifications to the operating system, which has obvious deployment drawbacks for using such jailing systems in large-scale distributed systems. Most jailing systems are intended to protect the operating system from tampering by untrusted programs. However, most jailing systems are not designed to prevent leakage of information from one process to another, in particular when those processes are executed by

[10] We currently use web URLs to store the data files for simplicity. Future implementations may store the information in protected rooms in the Mansion world, from which only pre-registered agents authenticated as representing the obtainer of the files can obtain the information.

[11] World and room access may be restricted to agents being owned by doctors or biological researchers, by having agents appropriately authenticated and co-signed (or resigned) at world entrance time. This is possible in Mansion; details are outside the scope of this paper.

[12] E.g., digital libraries generally only make abstracts of papers or books available to the public, in the hope that this provides sufficient information for the user to decide to buy such a book or paper. Confinement allows users to do full-text search or comparison without relying on abstracts or content owner-provided search engines to decide to buy a book (or, conceivably, music or a movie).

the same user. No system call interception based system to date has covered all avenues (e.g., using the semantics of some IPC related system calls) for transporting information from one process to another, and protecting against information flow through covert channels is generally not treated at all.

Much work has been done on adapting Java to the needs of mobile agents, to making Java more secure, and to adding resource management to Java [37, 20, 38]. Most of this work requires changing the JVM, which limits deployment of the described solutions on a large scale. Java-specific solutions limit applicability of those solutions for many applications, such as for mobile agents containing legacy code. No Java system to date deals with prevention of information leakage from one Java program (or thread) to another directly.

6. DISCUSSION

This paper describes an approach for building secure distributed systems using the concept of confined mobile agents, and demonstrated this approach in the context of the Mansion mobile agent system. The Mansion system provides the concept of a confined room, in which agents are automatically confined such that information flow from the room is controlled by a trusted guardian agent, which enforces an information flow policy defined by the room's owner.

Agent confinement allows for flexible searching through data collections with confidential or classified data or intellectual property while limiting the capabilities of a user to obtain information from these data collections. As agents execute locally, the information never leaves its host, except when explicitly allowed by the confinement policy defined by the data owner. This makes sure that the data owner stays in control over where or to whom his or her data is exported. Other than using, for example, watermarking, our system poses no constraints on the content of the data or its access mechanism. Rather, it works by using a mechanism which *prevents* data flow, rather than detect information leakage after the fact. To implement agent confinement securely, we designed a jailing system which is specifically intended to prevent information flow from one jailed process to another.

The examples of using mobile agents as described in this paper demonstrate a few important benefits of using mobile agents (or mobile code) confinement:

- Its algorithms can be customized to search for specific content of raw data.
- Agents access this data locally where the data resides, and thus can efficiently access this data, and
- Export of the data is controlled by the room (data) owner using the Guardian Agent, such that an agent can only export a limited amount of data or record identifiers from a room, depending on a policy enforced by the guardian agent.

The combination of these properties demonstrate that mobile agents are an important complement to existing technologies for building distributed systems. Mobile agents allow for fully customized search of (raw) data in order to select suitable datums which can then be exported from the confined room under conditions set by the data owner.

The Mansion paradigm provides a clear concept for storing and managing security sensitive data, the confined room. Confined rooms are marked to contain security sentitive data, and the Mansion middleware confines agents in a confined room in such a way that they can only export information from that room through the room's guardian agent. Agents can find a (confined) room by following (appropriately annotated) hyperlinks, and search for suitable information there. The data filtering mechanism is application specific and is fully embedded in the guardian agent's implementation, and can be adapted to the application's needs by the data owner.

For secure applications, careful design of the actual protocols is important. For example, for a medical application which allows agents to search through patient records, it is very important that the format of the internal data is sufficiently structured that the guardian agent can filter it correctly, to avoid that confidential patient information is exported accidentally. Such issues are to be resolved by an application designer; the Mansion system provides the necessary building blocks (such as the Mansion middleware, object server and jailing system, and the confined room concept), but cannot solve all issues that have to be considered for an end application to be secure. Future research is needed to design and analyze appropriate protocols for specific applications, such as sensitive medical data. We are currently working on the design of a guardian agent and its protocols in the context of a world containing genomic data files.

Building a confinement system is not trivial, as its security depends on secure confinement mechanisms (i.e., a jailer) and on a thorough evaluation on whether information from the room can be exported through weaknesses in the guardian agent's policies (which are generally application dependent). However, given that secure confinement mechanisms such as jailing develop further, and that policy specification and enforcement are given proper care, we believe that using confinement of mobile agents or mobile code is a feasible approach to achieve increased system security, and a valueable addition to existing instruments for building secure distributed systems.

Acknowledgements

The authors thank Bruno Crispo and the anonymous reviewers for useful comments on an earlier version of this paper. Ádám Balogh and Rutger Hofman are thanked for contributions to the jailing system described in this paper. Stichting NLnet is thanked for financial support.

7. REFERENCES

[1] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14(2), February 1997.

[2] M. van Steen; P. Homburg; A.S. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurrency*, January-March 1999.

[3] B.C. Popescu; M. van Steen; A.S. Tanenbaum. A Security Architecture for Object-Based Distributed Systems. *Proc. 18th IEEE Annual Computer Security Applications Conference*, December 2002. pp. 161-171.

[4] IBM. Web Services Security (WS-Security). 2002. http://www-106.ibm.com/developerworks/webservices/library/ws-secure/.

[5] I. Cox; J. Kilian; T. Leighton; T. Shamoon. Secure Spread Spectrum Watermarking for Multimedia. *IEEE Transactions on Image Processing vol. 6, no. 12*, pages 1673–1687, 1997.

[6] A.M. Eskicioglu; J. Town; E.J. Delp. Security of Digital Entertainment Content from Creation to Consumption. *Signal Processing: Image Communication, 18(4)*, pages 237–262, 2003.

[7] D. Milojicic; F. Douglis; R. Wheeler, eds. Mobility: processes, computers and agents. *ACM Press*, 1999.

[8] S.G. Belmon; B.S. Yee. Mobile agents and Intellectual property protection. *Rothermel and Hohl, eds. Proc. 2nd Int'l workshop on Mobile Agents (MA), LNCS 1477, Springer Verlag*, pages 172–182, 1998.

[9] G. Vigna (ed.). Mobile Agents and Security. *LNCS 1419*, 1998. Springer-Verlag.

[10] G.J. van 't Noordende; F.M.T. Brazier; A.S. Tanenbaum. Security in a Mobile Agent System. *1st IEEE Symposium on Multi-Agent Security and Survivability*, 2004. Philadelphia, PA.

[11] G.J. van 't Noordende; A. Balogh; R.F.H. Hofman; F.M.T. Brazier; A.S. Tanenbaum. A Secure and Portable Jailing System. *Technical Report IR-CS-025, Vrije Universiteit*, October 2006.

[12] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interception Based Security Tools. *Proc. Symposium on Network and Distributed System Security (NDSS)*, 2003. pp. 163-176.

[13] A. Fuggetta; G.P. Picco; G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering, vol. 24, no. 5*, pages 342–361, 1998.

[14] B. Walker; G. Popek; R. English; C. Kline; G. Thiel. The LOCUS distributed operating system. *Proc. 9th Symposium on Operating Systems Principles (SOSP)*, pages 49–70, November 1983.

[15] N. Suri; J.M. Bradshaw; M.R. Breedy; P.T. Groth; G.A. Hill; R. Jeffers. Strong mobility and fine-grained resource control in NOMADS. *Proc. Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA)*, pages 2–15, 2000.

[16] L. Bettini; R. De Nicola. Translating Strong Mobility into Weak Mobility. *Proc. 5th International Conference on Mobile Agents (MA)*, 2001.

[17] A.J. Chakravarti; X. Wang; J.O. Hallstrom; G. Baumgartner. Implementation of Strong Mobility for Multi-Threaded Agents in Java. *Proc. International Conference on Parallel Processing (ICPP)*, 2003.

[18] T. Sakamoto; T. Sekiguchi; A. Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. *Agent Systems, Mobile Agents, and Applications (LNCS 1882)*, pages 16–28, 2000.

[19] S. Funfrocken. Transparent Migration of Java-Based Mobile Agents: Capturing and Reestablishing the State of Java Programs. *Proc. 2nd International Workshop on Mobile Agents (MA)*, pages 26–37, september 1998.

[20] W. Binder; J.G. Hulaas; A. Villazón. Portable Resource Control in Java - The J-SEAL2 Approach. *Proc. 16th. ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2001.

[21] V. Roth; U. Pinsdorf; J. Peters. A Distributed Content-Based Search Engine Based on Mobile Code. *Proceedings of the 2005 ACM symposium on Applied computing (session: Agents, interactions, mobility and systems (AIMS)), New Mexico*, pages 66–73, 2005.

[22] J.E. White. Telescript Technology: Mobile Agents. *White paper, General Magic*, 1996.

[23] J. Baumann; F. Hohl; M. Strasser; K. Rothermel. Mole - Concepts of a Mobile Agent System. *Technical Report, Universität Stuttgart*, August 1997.

[24] N. Karnik and A. Tripathi. Security in the Ajanta Mobile Agent System. *Software - Practice and Experience 31(4)*, 2001. pp. 301-329.

[25] D. Lange and M. Othima. Mobile Agents with Java: The Aglet API. *World Wide Web 1(3)*, September 1998.

[26] R.S. Gray; D. Kotz; G. Cybenko; D. Rus. D'Agents: Security in a Multiple-language, Mobile-agent System. *Mobile Agents and Security*, 1998. LNCS 1419, Springer-Verlag pp. 154-187.

[27] H. Peine and T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. *Proc. First Int'l Workshop on Mobile Agents*, 1997. LNCS 1219, Springer-Verlag.

[28] D. Johansen; R. van Renesse; F.B. Schneider. Operating systems support for mobile agents. *5th Workshop on Hot Topics in Operating Systems*, 1995. pp. 42-45.

[29] N.J.E. Wijngaards; B.J. Overeinder; M. van Steen; F.M.T. Brazier. Supporting Internet-Scale Multi-Agent Systems. *Data and Knowledge Engineering 41(2-3)*, 2002. pp. 229-245.

[30] V. Roth; M. Jalali-Sohi. Concepts and Architecture of a Security-Centric Mobile Agent Server. *Proc. 5th International Symposium on Autonomous Decentralized Systems (ISADS)*, page 435, 2001.

[31] N. Provos. Improving Host Security with System Call Policies. *Proc. 12th USENIX Security Symposium*, August 2003. pp. 257-272.

[32] T. Garfinkel; B. Pfaff; M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. *Proc. ISOC Network and Distributed System Security Symposium (NDSS)*, 2004. .

[33] I. Goldberg; D. Wagner; R. Thomas; E.A. Brewer. A Secure Environment for Untrusted Helper Applications - Confining the Wily Hacker. *Proc. 6th Usenix Security Symposium*, 1996. San Jose, CA, USA.

[34] K. Jain; R. Sekar. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion detection and Confinement. *ISOC Network and Distributed System Security Symposium (NDSS)*, 2000. pp. 19-34.

[35] T. Shinagawa; K. Kono; T. Masuda. Flexible and Efficient Sandboxing Based on Fine-Grained Protection Domains. *ISSS*, 2002. pp. 172-184.

[36] D.S. Peterson; M. Bishop; R. Pandey. A Flexible Containment Mechanism for Executing Untrusted Code. *Usenix Security Symposium*, 2002.

[37] W. Binder and V. Roth. Secure mobile agent systems using Java: where are we heading? *Proceedings of the 2002 ACM Symposium on Applied Computing*, 2002. pp. 115-119.

[38] G. Back; W.C. Hsieh; J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. *Proc. 4th Symposium on Operating Systems Design and Implementation (OSDI), San Diego, CA.*, pages 333–346, October 2000.