A Secure Modular Mobile Agent System

Adam Pridgen
Christine Julien

TR-UTEDGE-2006-003

A Secure Modular Mobile Agent System

Adam Pridgen and Christine Julien
The Center for Excellence in Distributed Global Environments
The Department of Electrical and Computer Engineering
The University of Texas at Austin
{atpridgen, c.julien}@mail.utexas.edu

ABSTRACT

Applications in mobile multi-agent systems require a high degree of confidence that code that runs inside the system will not be malicious and that any agents which are malicious can be identified and contained. Since the inception of mobile agents, this threat has been addressed using a multitude of techniques, but many of these implementations have only addressed concerns from the position of either the platform or the agent, and very few approaches have attempted to approach the problem of mobile agent security from both perspectives simultaneously. Furthermore, no middleware exists that facilitates provision of the required security qualities of mobile agent software while extensively focusing on easing the software development burden. In this paper, we introduce a middleware system that eases the integration of core software assurance qualities into the mobile agent software development process by addressing perspectives from positions the of both the platform and the agent.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: [Computer-aided software engineering]; K.5 [**Security and Protection**]: []; K.6.3 [**Software Management**]: [Software development]

General Terms

Design, Security, Standardization

Keywords

Mobile Agents

1. INTRODUCTION

Mobile agent systems and applications are becoming highly prominent for tasks such as information sharing, analysis, evaluation, and response. Before these systems fulfill their promise, they must overcome the challenging obstacles of effective software development and security [12]. In general, agents are regarded as highly autonomous processes which can perform tasks ranging from simple queries to complex computations. In all cases, a major component of a mobile agent system is the platform that hosts the agents.

Motivating applications for mobile agents range across many domains, and the benefit of using mobile agents can far outweigh traditional approaches. In *epidemic updates*, agents carry firmware upgrades to remote clients by intelligently propagating through a network. As computer systems become increasingly pervasive, an effective means for propagating updates to these systems will be required, especially when physical contact with a device is infeasible. Epidemic updates also provide relief to manufacturers who require their vehicles to upgrade to a new firmware version without instantiating a massive recall. A dealership could strategically deploy updates on a few vehicles, and, when vehicles stop in public areas such as a market, the update can spread to nearby systems to which it applies.

Another distributed information application that could benefit from secure mobile agents focuses on collecting and comparing events in a distributed surveillance sensor network. In this system, agents can be utilized to help monitor arrays of many sensors like cameras, acoustics, and pressure sensors. Furthermore, these systems could be integrated look for elaborate acts of sabotage. The network monitoring agents could even be leveraged as a channel for comparing network-wide events among different organizations without disclosing sensitive or proprietary data.

As mobile agent deployments become more prominent and sophisticated, programming mobile agent components and applications needs to be simplified. A modular mobile agent platform could help capture this desired flexibility for easy integration, but the system must also incorporate trusted computing functions and must facilitate the integration of future technologies. A simple middleware that permits domain programmers to develop these components and allows expert programmers to extend the system is essential. Since mobile agents and their respective applications have a significant degree of variance, strategies such as cross-layer design should be employed in many situations.

In this paper, we introduce the Secure Modular Mobile Agent System.H (SMASH), which provides modularity for agent and platform components, information assurances, and mechanisms to assist mobile agents as they move between platforms. SMASH is also designed to address context-based agent execution and security, enable coordination among agents and platforms, and, overall, improve programmablity, security, and extensibility for highly versatile mobile agent applications. SMASH seeks to allow a wider range of authentication methods, rather than restrict agent authentication to code signing as employed in Java-based approaches. To support unpredictable travel patterns, SMASH supports strict authorization and resource control measures yet eliminates the burden of excessive authentication for transient agents as they move to their destinations.

The rest of this paper is organized as follows. Section 2 reviews previous research Section 3 provides an overview of our architecture. Details about implementation and the programming interface will be further discussed in Section 4. Future work is in Section 5, and Section 6 concludes.

2. RELATED WORK

Information assurance for mobile agents is a daunting task because security threats arise from agents attacking other agents or platforms and from platforms attacking agents. The ultimate challenge is to manage trust between components of the agent system. Providing middleware for such systems is non-trivial because it must forecast and abstract implications which may arise in the various roles and actions of remote agents and platforms. Issues such as software exceptions, resource availability, etc., can open subtle holes for exploitation or even cause a system to fail.

In the area of software assurance, a number of projects have increased the probability of dynamically detecting data or code tampering. One such framework [6] re-arranges code at compile time to obtain a unique binary and then embeds a unique watermark created from standard encryption algorithms. This dramatically suppresses the ability of an adversary to manipulate any portion of the code and can also be useful in maintaining a light-weight agent.

Page et. al. [11], explore a method in which each agent performs a randomly periodic self-examination to ensure no modifications have been made while the agent was executing. Other methods use reference states [5], state appraisals [2], and even agent execution traces [18]. These methods can add weight to the agent code and payload, require *a priori* knowledge or consistent connectedness of platforms for verification, and, under some circumstances, data appended to the agent can be forged.

Most mobile agent systems have been built on Java or varying scripting languages. Projects using Java utilize the JVM's Security Manager, but this management system can be intractable due to an excessive number of security policies and unscalable as mobile agent systems become more complex. On the flip side, Java offers more robust, object-oriented programming, an elaborate API library, and portable code. Mobile agent systems implemented in scripting languages are also portable and have stronger mobility, but they do not provide extensive security management and they tend not to be as object-oriented.

There are a number of middleware projects for secure mobile agents, and we sample only a small fraction of them here. MARS [1] explicitly separates the data an agent can access from the host's file system through a novel coordination approach, but this reduces flexibility and requires significant *a priori* knowledge to populate the agent accessible data space. In addition, MARS is dramatically limited in the granularity of access control it can provide. Nomads [15] implements many promising features such as fine-grained access control, strong mobility, and flexible execution profiles based on application and context; however, Nomad agents run in a custom execution environment that dramatically reduces the code portability of the agents. D'Agents [4] supports multiple agent implementation languages and also differentiates anonymous and authenticated agents. Aglets [8] are applet agents based on Javascript. They provide conditional access rights and moderate access control based on aglet "identity."

Ajanta [9] is another mobile agent system built on Java, that implements extensive access control measures, rather than relying entirely on Java's Security Manager. Ajanta suffers due to Java's constrained policy system. Ajanta introduces containers for appending read-only data and a stronger security manager that controls access to resources by requiring agents to set-up resource proxies that access resources through the manager as established by the platform's policies. In an effort to make agents lightweight, each agent carries an address for a trusted code server, from which it can dynamically load supplemental Java classes.

Java has been a very important tool in the mobile agent community. Java's portability, type safety, automated memory management, serialization, and built-in security management have made it the language of choice for many developers. However, for the purposes of strict information assurance, Java has fundamental inadequacies. For example, the JVM is not intended as a multi-user execution environment, so a Java-based mobile agent system has limited ability to govern all resources of agents and threads [10]. A second issue with Java-based systems is that they were meant typically for on-platform management in which an agent derives its platform access rights from those established locally on the platform. There is no method for the platform to dynamically check access policies within a local domain. Also, because access controls are issued per domain, either each visiting agent must have its own domain or agents must share domain privileges. The former is unscalable and unfriendly to open systems. The latter neglects *The Principle of Least Privilege* allowing dissimilar agents to have the same permissions even when those privilege are unnecessary [14]. Additionally, Java cannot authorize access based on a particular task or goal, dramatically restricting the potential for context-based authorization and privileges.

SMASH strives to enhance the software engineering of mobile agents by introducing a modular and adaptable system, so application developers can quickly customize mobile agents and platforms to their needed specifications and security requirements. SMASH emphasizes *security by design* but provides modularity so future application designers do not need to design around the architecture, but rather design for their application.

3. SMASH: AN OVERVIEW

In this section we introduce our middleware, the Secure Modular Mobile Agent System.H (SMASH), which facil-

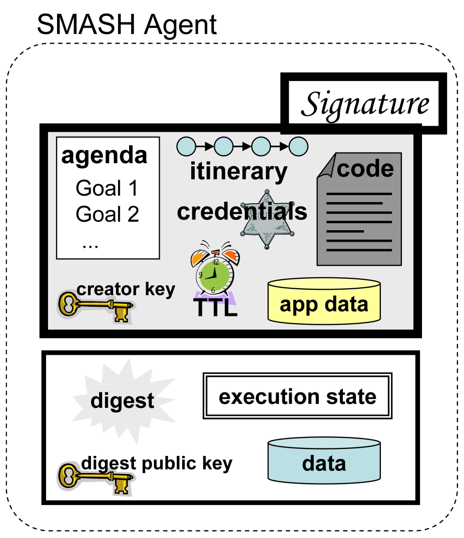

Figure 1: Depiction of a Mobile Agent in SMASH

itates openness, security, and modularity in mobile agent systems. SMASH fosters openness by allowing anonymous mobile agents to move freely through platforms while still coordinating with other agents. Additionally, SMASH enables coordination among these anonymous entities. Security is provided for more robust agents by integrating hardware and software assurance methods to prevent tampering, enhance authentication, and to authorize agents and platforms through traditional and contextual means. SMASH's core design principle is modularity, which improves mobility, eases the programming effort, and allows customization. Thus, SMASH focuses on adapting the system to an application *not* an application to the system.

3.1 SMASH Agents and Functionality

To adhere to an open architecture, SMASH supports two types of agents, *anonymous* and *authenticated* agents, in a manner similar to [4]. An anonymous agent is simply one that has not authenticated with the platform. When an agent is unauthenticated, its functionality on the platform is restrained. Such an agent may access only designated read-only data, read and write to a Blackboard, perform simple computations, or leave. This anonymous classification allows agents to move through intermediate platforms without having to authenticate with each of them, which can improve performance and reduce the latency caused by unnecessary authentication. An authenticated agent, on the other hand, is one that has sufficiently proven its identity to the platform. An agent's identity refers to with whom the agent is associated (e.g. user, group, platform, etc.). Once an agent's identity is verified, it is awarded rights based on that identity and/or the context of its task(s).

Figure 1 represents the components of a SMASH agent. All agents are composed of modules, which are simply defined architectural types, methods, and functions of the mobile agent. SMASH agents contain an immutable *main* module shown at the top of the figure. This main module comprises the following submodules: an agenda, code, credentials, ap-

```
<Goal>
  <Task>
    <Upgrade>
      <attribute>
   Description = "Upgrade Firmware"
      </attribute>
      <type=Firmware>
        <attribute> Version = "3.0 Beta" </attribute>
        <attribute> PreviousVersion = "2.99" </attribute>
        <attribute> SystemArch=any </attribute>
        <attribute> ApplyAt="12:00a" </attribute>
        <attribute> EstimatedTime="3 minutes" </attribute>
      </Firmware>
    </Upgrade>
  <Task>
  <Resources>
    <Internal>
      <attribute> UpgradeData =  "./Firmware3.bz2"</attribute>
</Internal>
  <Network>
      <attribute> BandwidthRate = 10Kb </attribute>
      <attribute> BandwidthUsage = 5MB </attribute>
      <attribute> PortUsed = 80 </attribute>
      <attribute> PortHandle = "NHandle" </attribute>
<attribute> RemoteHost = "foo.bar.org" </attribute>
    </Network>
    <WorkingDirectory>
      <attribute> Directory = "" </attribute>
      <attribute> FileAccessRights = "rwx" </attribute>
      <attribute> DirectoryHandle = "FHandle" </attribute>
    </WorkingDirectory>
  </Resources>
  <AuthorizationRequirements>
    <attribute> User = "operator" </attribute>
    <attribute> Directory = "/usr/bin" </attribute>
    <attribute> FileAccessRights = "rw" </attribute>
  </AuthorizationRequirements>
</Goal>
```

Figure 2: Model of an Update Goal

plication data, itinerary, and a time-to-live. The agent's agenda contains the agent's goals, described in more detail in Figure 2, which shows the goal definition for an agent performing a firmware upgrade. There are at least three required fields for any goal: the Task, Resources, and AuthorizationRequirements. The Task is functionality of the specified goal, and the Resources list what resources the agent needs to use to complete the task, which may consist of system resources, remote resources or data, and even external functions. Finally, the AuthorizationRequirements provide a recommended set of privileges to accomplish the task; however the final determination of actual privileges is left to the platform. The code is the executable portion of the agent, and the application data contains static data accessed throughout the agent's execution.

The itinerary submodule contains the agent's travel plan and designates the platforms the agent plans to visit. An itinerary includes several components associated with each target platform. First, if the agent intends to use privileged services, the itinerary includes a checksum of the expected software the platform should be running. Second, the itinerary also includes information unique to each platform, so the agent can authenticate with it. Finally, the itinerary designates whether the agent will allow a particular platform to refer it to another *trusted* platform. A referral occurs if a platform does not have a resource, but knows of another platform with the agent's required resources. If the agent accepts referrals, then the agent will go to the referred platform and honor the trust relationship between the platforms; this relies on a third party authentication service that enables the agent and platform to verify each other's identity.

The final submodule in the agent's main module contains the agent's credentials. This component provides a *tamper detector* comprising a public key, a signature, and the agent's prescribed authentication methods, *authentication submodules*, which describe how an agent can authenticate with platforms it encounters. Each authentication sub-module contains the information necessary to complete the prescribed authentication method. The platform must support at least one of an agent's prescribed methods, or the two will not be able to mutually authenticate.

The tamper detector protects all sub-modules within the main module. When the agent is completely assembled and prepared for dispatch, the creating entity will select a private key to use to sign the agent. Before signing the main module, the creator will place the public key into the credentials and identify the asymmetric algorithm being used. The agent then signs the main module using the selected private key.

In addition, an agent may contain an additional dynamic module, shown in the lower portion of the agent in Figure 1. This module stores vital state and process data with a high degree of confidence as the agent moves from platform to platform. The execution state includes information about the variables in memory and the instruction where the agent left off on the previous platform. The data refers to any computation results, accumulation of logs, etc., that the agent generates throughout its tasks and wishes to maintain. An effort is made to protect this data from tampering using a digest. Before departing a platform, the agent creates a hash of the execution state and application data (using a function like MD-5) and passes this hash to the platform. The platform signs the combination of the hash and the platform's public key. The agent receives the signed hash and the public key from the platform, which it stores as a digest and the digest public key. When the agent initializes on a new platform, it verifies the data and state information using the reverse process. The public key is also matched against the public key of the previous platform in the agent's itinerary. If a the key cannot be found or the digest is wrong, the agent will self-destruct.

3.2 SMASH Platform and Functionality

Like SMASH agents, the host platform is engineered to provide support for an open architecture, extensibility, and various levels of security. As shown in Figure 3, we use a layered approach to compartmentalize our architecture's functionality and to prevent an outbreak of malicious activity. At the lowest level of the architecture, the operating system handles issues like communication, system level access controls, etc. When an agent arrives at the platform, an integrity check is administered, and, upon successful completion, the agent is moved to the Untrusted Layer. The agent then moves through the Authentication and Authorization Layer, where the agent and platform mutually authenticate to become a trusted entity. After successful authentication and

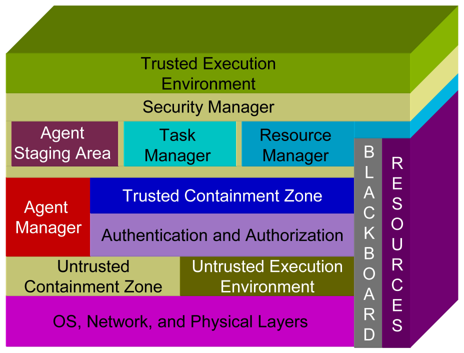
Trusted Execution Environment
Security Manager
Agent Staging Area
Task Manager
Resource Manager
Agent Manager
Trusted Containment Zone
Authentication and Authorization
Untrusted Containment Zone
Untrusted Execution Environment
OS, Network, and Physical Layers
BLACKBOARD
RESOURCES

Figure 3: SMASH Platform Architecture

authorization, the agent is placed into the Trusted Containment Zone (TCZ). From here up, the agent will interact directly with the Security Manager to obtain the required resources and be executed. Within the Security Manager, The Task Manager determines if the platform can provide useful services to the agent (based on the agent's goals), and the Resource Manager sets-up proxies so the agent can access resources external to the execution environment. The Agent Manager tracks all agents on the platform.

SMASH's final components, shown to the right in Figure 3, represent publicly accessible platform resources. The Blackboard is a memory-constrained FIFO queue that is available to any agent (authenticated or anonymous) for read-write access. Agents can use this space to coordinate tasks. Since agents are kept completely isolated, this is one of two ways that they may interact with each other; the other method will be discussed later. This data space also allows agents to mark a platform as visited. The platform also has the ability to make other parts of memory public and read-only (similar to a glass-enclosed bulletin board).

Figure 4 shows the entire process of admitting an agent to a platform. When an agent first arrives at the platform the agent and platform perform initial integrity checks. The platform will use the *tamper detector* located in the agent's credentials discussed earlier. The agent queries a special hardware chip that provides the agent with a checksum of the currently running system software. The agent compares the hardware module's result with the one stored in the agent's itinerary. If they match, the platform's check has passed and the agent will register with the Agent Manager (AM) as an *anonymous agent* and proceed to the Untrusted Containment Zone (UCZ).

The agent receives few privileges in the UCZ. Here the agent has limited processing power, may read and write to the platform's Blackboard, has access to the platform's read-only memory, or leave the platform. The agent may also piggyback on the platform to get to some physical location. When the agent is registered with the Agent Manager (AM), it is scheduled to receive minimal processing time in a low-priority queue. The AM also monitors agents, and, if they die unexpectedly, removes them from the UCZ. If an agent simply needs to obtain some public data from the platform,

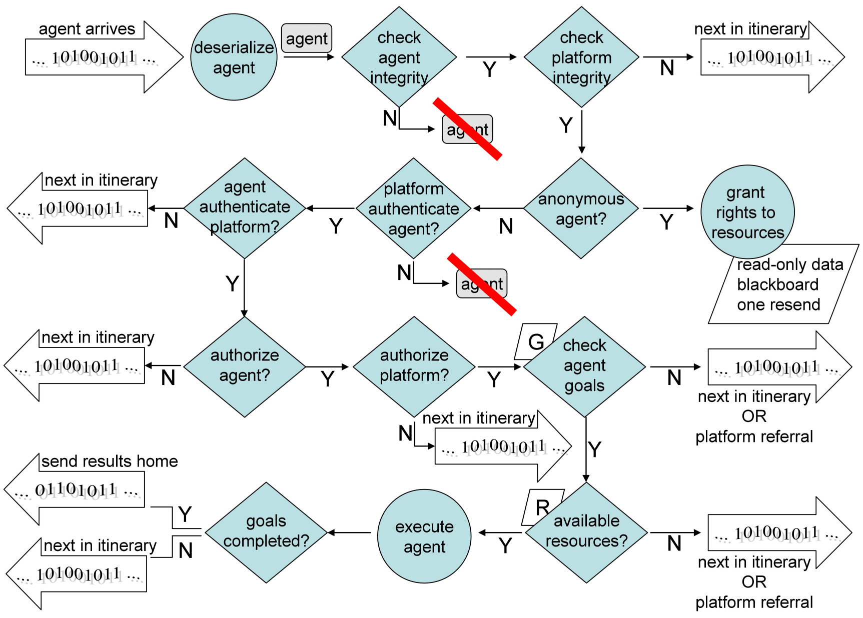

Figure 4: Decision Tree Used by the SMASH Platform.

it can use its processing time to query and then leave. On the other hand an agent may also use this processing time to inform the AM that it wishes to authenticate with the platform.

After the agent signals the AM that it wishes to authenticate, the AM moves the agent into the Authentication and Authorization Layer (AAL). In the AAL, the agent and the platform mutually authenticate each other. The platform queries the agent about how it can authenticate, and the agent does the same for the platform. If the two possess some method necessary of authentication in common, then they can mutually authenticate. If this is not the case the agent is removed from the AAL and flagged in the AM, meaning it can no longer attempt to authenticate. If mutual authentication succeeds an authorization service is launched. The authorization service will look locally, to a remote server, and even employ another mobile agent service [13] to grant the agent access privileges. The authorization source is platform-dependent, but it must establish whether the agent can use the platform, at what privilege level, and which resources should be accessible. The agent can leverage the same services to authorize the platform to ensure no revocations have taken place since it was dispatched. Once these authorizations complete, the status of the agent is updated to *authenticated* in AM, and the agent is moved into the Trusted Containment Zone (TCZ).

After the agent is given an initial set of privileges, it passes its agenda to the Security Manager (SM). From here, the agent will interact only with the SM. The SM passes the agenda to the Task Manager (TM), which analyzes the agenda, the agent's privileges, and which of the agent's tasks are currently permissible. If the TM sees a possible match, (i.e. some task is permissible and requires equal or less access than the agent's currently assigned privileges), the TM passes the agent's requested resource list to the Resource Manager (RM), which locates the desired resources and intiates proxies for the agent to use to access those resources. The RM adheres to the order in which resources are required, if the agent provides such information. This expedites the agent's execution, reduces idle time, and helps release resources in a timely manner for other agents running on the platform.

When the necessary resources become available, the agent is moved into the Agent Staging Area (ASA), and the status of the agent is updated in the AM. In the ASA, the agent's Bootstrap Code (BC) is identified and loaded into the staging area. The BC first goes through all of the agent's modules to ensure no tampering or corruption has occurred in the agent's immutable sections. The BC then loads the agent into memory. The agent checks all execution environment parameters such as handles and variables and initializes them appropriately for this platform. This allows an agent to resume execution at the point it previously left off. If any failure occurs, the BC aborts and self-destructs or returns home. Finally, the BC updates the agents status with the AM to *executing*.

While the agent executes, the SM monitors the agent for any deviant behavior like excessive bandwidth usage or attempts to access restricted resources. Depending on the severity of the violation, the SM can kill the agent, make the agent leave, downgrade the agent, or throttle the agent. When the agent's execution ends, the BC moves the agent back to the agent staging area. Here, the BC checks the agent's integrity and inventories the modules. The BC will obtain a digital signature for the data and execution state (the digest). After the BC completes the clean-up, it will signal to the AM its intention to leave, and the AM will provide it with a means to leave the platform.

4. SMASH INTERNALS

SMASH eases programming of secure mobile agent applications by providing a modular and extensible programming framework for *anonymous* and *authenticated* agents. SMASH also provides the support necessary for these agents to authenticate and validate the platforms they visit and vice versa.

4.1 SMASH Implementation Details

In this section, we explore aspects of the implementation details of the SMASH middleware. To enable several aspects of tamper detection in SMASH, our implementation utilizes Security Enhanced Linux (SELinux [16]) which is a Linux kernel modified and partly maintained by the National Security Agency. SELinux enables granular access controls and provides a powerful but securable multi-user environment. In addition, we require each platform to incorporate a Trusted Platform Module (TPM), a hardware chip specified by the Trusted Computing Group [17]. In combination, this hardware and operating system enable our implementation of the security and trust mechanisms outlined in the previous section.

As described in Section 2, we avoid using the Java programming language for creating agent systems due to its many restrictions in securing application interactions. Instead, our middleware is implemented on the platform described above using C++. On top of this, we use Python to provide a more flexible and easy-to-use programming interface to the application developer. The next section describes this interface in a bit more detail. Python is a powerful object-oriented language whose features make it attractive for rapid application development. C++, on the other hand, is more amenable to interaction with SELinux operating system services, has better performance, and makes many of the subtle aspects of the system described in Section 3 possible. Finally, SMASH assumes network communication to be handled by the operating system, and the middleware simply handles agent movement between platforms at the application level.

4.2 SMASH Application Details

Because our goal is to simplify the process of programming secure mobile agents, SMASH provides much of the agent's reusable functionality within the middleware. The programming interface through which developers create SMASH agents is presented in Python, an intuitive object-oriented language. Specifically, the middleware provides an agent base class (`agent`) that any application agent must derive. This base class contains an `__init__` method to which the deriving agent can provide the aspects of the *main* module depicted in Figure 1. Each of these submodules *except the agent's code* (i.e., the agenda, the itinerary, the credentials, the creator key, the TTL, and the application data) are represented by additional Python classes in the SMASH middleware that the developer can instantiate. Ultimately, when an application developer's agent is created, its `__init__` method is invoked, and, within this method, the submodule components are either received as parameters or created, and the method `agent.__init__( ...)` is called with the submodule components as parameters. This successfully initializes an agent.

One thing worth noting about this organization of the programming interface is the ease with which the submodules can be initialized. Take as an example the XML-style definition of goals (shown in Figure 2). To initialize its agenda, the agent needs only to pass the XML file(s) defining the agenda to the Python `agenda` class, and the mechanics for parsing and properly storing the details of the goals are implemented within the middleware. The agent's itinerary (which includes not only the ids of the target platforms but also keys and other cryptographic information used for agent admission) is also defined via a standard XML format that can be automatically processed by the middleware. Similar standard approaches for representing the other submodules module are used; details are omitted here for brevity.

SMASH is engineered to provide both strong and weak mobility. As such, the `agent` base class in the middleware contains two methods; an agent overrides one or the other depending on whether it desires strong or weak mobility. In addition, the deriving agent sets a flag in the base class indicating its selection. The two methods, `strong_run` and `weak_run` have no functionality defined in the base class, but in the former case, when the derived agent decides it is time to move to a new platform, the exact execution state is saved and later restored on the new platform. This means that the agent actually records how much processing has occurred and restarts itself on the new platform in exactly that location. In the case of weak mobility, when the agent moves, its `weak_run` method simply restarts from the beginning. To move, a derived agent calls the `move` method in the `agent` base class, which first determines which mobility method is being used and (if necessary) saves the agent's execution state. Then the `move` method hooks in to the remainder of the middleware to find the next platform in the itinerary and move there. Functionality implemented within the middleware handles the admission of the agent as described in Section 3, and eventually, if all checks pass successfully, the Python derived agent class is restarted.

5. FUTURE WORK

The SMASH middleware infrastructure as described in Section 4 is currently under development. Future work will see the completion of the middelware as described here and the development of several agent-based applications using the middleware to demonstrate not only the range of security concerns the middleware covers but also the range of application domains in which it can be applied. This evaluation will be an important point in the process of SMASH development because we hope to gain insight into the languages and representations we have chosen for modules and submodules (e.g., the representation of the agent's tasks and resources using XML).

At the model level, SMASH currently incorporates context information into security decisions by accounting for an agent's intended tasks and requested resources. In the future, we plan to extend this framework to incorporate the use of more context information into security processes. This work will build on our previous work in constructing context-sensitive access controls [7]. We envision that context information such as the other agents present on a host, available resources, network connectivity, and quality of service will be able to positively influence agents' and platforms' admission and authorization decisions to provide efficient behavior to the platforms and the best possible level of service to the mobile agents.

6. CONCLUSION

In this paper, we have presented SMASH, a secure mobile agent middleware that builds on past research efforts to create a secure, open, and modular mobile agent system that can be used to experiment and develop robust applications. SMASH embraces openess by directly considering agents' goals and requests in the security process, allowing anonymous agents to move freely through intermediary platforms onto their final destinations, and providing a means for agents to coordinate or compute without being authenticated to a platfrom. Security is provided by integrating hardware and software techniques to protect, detect, and identify tampering, and to a high degree, provide increased flexibility with respect to agent authentication and authorization. Finally, SMASH's design modularity enhances the extensibility of mobile agent applications, eases the development burden associated with mobile agent systems, and empowers developers with tools to create a new generation of mobile agent applications.

7. REFERENCES

[1] G. Cabri, L. Leonardi, and F. Zambonelli. Mars: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, 2000.

[2] W. Farmer, J. Guttman, and V. Swarup. Security for Mobile Agents: Authentication and State Appraisal. In *Proc. of the 4th European Symp. on Research in Computer Security*, pages 118–130, Springer-Verlag, September 1996.

[3] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proc. of the 25th IEEE Int'l. Conf. on Distributed Computing Systems*, pages 653–662, 2005.

[4] R. S. Gray, D. Kotz, G. Cybenko, and D. Rus. D'agents: Security in a multiple-language, mobile-agent system. In *Mobile Agents and Security*, pages 154–187, London, UK, 1998. Springer-Verlag.

[5] F. Hohl. A Framework to Protect Mobile Agents by Using Reference States. *Proc. of the 20th IEEE Int'l. Conf. on Distributed Computing Systems*, pages 410–419, 2000.

[6] M. Jochen, L. Marvel, and L. Pollock. A Framework for Tamper Detection Marking of Mobile Applications. In *Proc. of the 14th Int'l. Symp. on Software Reliability Engineering*, pages 143–152, 2003.

[7] C. Julien, J. Payton, and G.-C. Roman. Adaptive access control in coordination-based mobile agent systems. In *Software Engineering for Large-Scale Multi-Agent Systems III*, volume 3390 of *LNCS*, pages 254–271, February 2005.

[8] G. Karjoth, D. B. Lange, and M. Oshima. A security model for aglets. *IEEE Internet Computing*, 1(4):68–77, 1997.

[9] N. M. Karnik and A. R. Tripathi. Security in the Ajanta mobile agent system. *Software—Practice and Experience*, 31(4):301–329, 2001.

[10] P. Marques, N. Santos, L. Silva, and J. G. Silva. The security architecture of the M&M mobile agent framework. In *Proc. of the SPIE's Int'l. Symp. on The Convergence of Information Technologies and Communications*, 2001.

[11] J. Page, A. Zaslavsky, and M. Indrawan. Countering Security Vulnerabilities in Agent Execution Using a Self Executing Security Examination. *Proc. of the 3rd Int'l Joint Conf. on Autonomous Agents and Multiagent Systems*, pages 1486–1487, 2004.

[12] V. Roth. Obstacles to the Adoption of Mobile Agents. In *Proc. of the IEEE Int'l. Conf. on Mobile Data Management*, pages 296–297, January 2004.

[13] A. Seleznyov, M. O. Ahmed, and S. Hailes. Agent-based middleware architecture for distributed access control. In *Proc. of the 22nd Int'l. Multi-Conf. on Applied Informatics: Articial Intelligence and Applications*, pages 200–205, 2004.

[14] Sun Microsystems. The Java 2 Platform. `http://java.sun.com/j2se`, January 2006.

[15] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, R. Jeffers, T. S. Mitrovich, B. R. Pouliot, and D. S. Smith. NOMADS: toward a strong and safe mobile agent system. In *Proc. of the 4th Int'l. Conf. on Autonomous agents*, pages 163–164, 2000.

[16] The SELinux Project. `http://selinux.sourceforge.net/`, December 2005.

[17] Trusted Computing Group. Trusted Computing Group Hompage. `https://www.trustedcomputinggroup.org/home`, November 2005.

[18] G. Vigna. Cryptographic Traces for Mobile Agents. In *Mobile Agents and Security*, volume 1419 of *LNCS State-of-the-Art Survey*, pages 137–153. Springer-Verlag, June 1998.