SMASH: Modular Security for Mobile Agents

Adam Pridgen
Christine Julien

TR-UTEDGE-2006-007

SMASH: Modular Security for Mobile Agents

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

Mobile agent systems of the future will be used for secure information delivery and retrieval, off-line searching and purchasing, and even system software updates. As part of such applications, agent and platform integrity must be maintained, confidentiality between agents and the intended platform parties must be preserved, and accountability of agents and their platform counterparts must be stringent. SMASH, Secure Modular Mobile Agent System.H, is an agent system designed using modular components that allow agents to be easily constructed and the system to be easily extended. To facilitate security functionality, the SMASH platform incorporates existing hardware and software security solutions to provide access control, accountability, and integrity. Agents are further protected using a series of standard cryptographic functions. While SMASH promotes high assurance applications, the system also promotes an open network environment, permitting agents to move freely among the platforms and execute unprivileged actions without authenticating. In the remainder of this paper, we elaborate on the components and capabilities of SMASH and present an application that benefits from each of these elements.

1 Introduction

Mobile agent systems and applications are becoming highly prominent for tasks such as information sharing, analysis, evaluation, and response, but before these systems can be fully utilized, security mechanisms in these services must be improved [1]. In general, software *agents* are regarded as highly autonomous processes which can perform tasks ranging from simple queries to complex computations. The counterpart to this system is the platform, which loads and executes the agent. *Mobile agents* augment the traditional agent's autonomy with the ability to move from platform to platform to accomplish their tasks.

Motivating applications for mobile agents range across many domains, and the benefit of using mobile agents in distributed computing applications can far outweigh traditional approaches. In *epidemic updates*, mobile 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 (e.g., when devices are located in remote regions or hostile environments). Epidemic updates also provide relief to automobile manufacturers or fleet maintainers who could upgrade their vehicles'

software without initiating 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 various sensors such as cameras, acoustics, and pressure sensors. Furthermore, all of these sensors could be used to create intelligent detection networks that combine and look for anomalies calculated from sensor readings combined across regions of the network. Network monitoring agents could even be used for collecting network-wide events and pushing the filtering and processing of these events into the network without disclosing sensitive or proprietary information.

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 enable coordination among agents and platforms, address context-based agent execution and security that enables adaptive services, and, overall, improve programmability, security, and extensibility for highly versatile mobile agent applications. SMASH utilizes asymmetric and symmetric cryptographic functions from existing encryption libraries, permitting more flexible authentication for both the agent and the platform, rather than restricting 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 will discuss the agent's components and functionality, and Section 3 will elaborate on the supporting platform's architecture and capability. In Section 4, properties of a secure system are discussed, and these qualities are related to SMASH's capabilities and design. Section 5 provides some example applications that can benefit from SMASH's architecture. Section 6 will discuss past work related to SMASH, while Section 7 concludes the paper.

2 Agent Components and Security Measures

This section will describe the SMASH agent architecture and then discuss how these components supply adequate security functionality. We start with an overview of the SMASH agent model and conclude with system and implementation details that provide a close inspection of a SMASH agent's inner workings.

2.1 The SMASH Agent Model

To adhere to an open architecture, SMASH supports two types of agents, *anonymous* and *authenticated* agents, in a manner similar to [2]. An anonymous agent is simply one that has not authenticated with the platform on which it is currently

located. Such an agent may access designated read-only data, read and write to a Blackboard, perform simple unprivileged actions, 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 and may represent a user, group, platform, application, etc. Once a platform verifies an agent's identity, the platform awards the agent rights based on its identity and/or the context of its task(s).

Fig. 1 shows a pictorial representation of the components found in any SMASH agent. All agents are composed of modules, which are simply defined architectural types, methods, and functions of the mobile agent. By taking this design approach, SMASH can take advantage of this component model and protect *pieces* of the agent in different ways, rather than protecting an entire agent in a single manner. In addition, this design supports modular development and evolution of agents, which reduces the development burden and may even enhance the ability to spawn new agents in an automated and consistent manner. SMASH agents contain an immutable *main* module shown at the top of the figure (with the darkened rectangular border). This main module comprises the following submodules: code, application data, agenda, itinerary, credentials, and a time-to-live. The main module is signed by the agent's creator, and this signature helps protect the agent's main module from unauthorized modification.

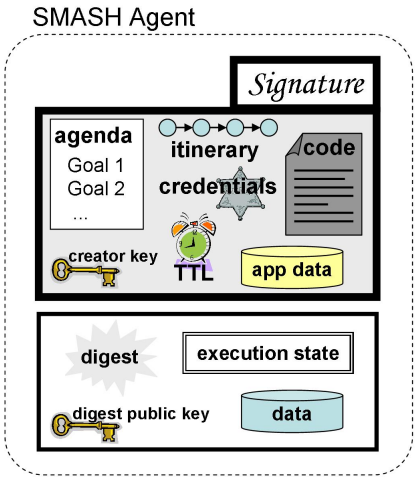

Fig. 1. A Mobile Agent in SMASH

The *code* shown in Fig. 1 is the executable portion of the SMASH agent, and the *application data* ("app data" in the figure) contains static data the agent carries during its travels. The application data is simply constant data that does not change throughout the execution lifetime of the agent. If the agent does need to modify this data, this modified data is placed in a secondary module ("data" in the figure), described in more detail below. The *TTL*, or time-to-live, is a time metric for specifying the lifetime of an agent. Since agents may get lost or the lifetime of data may expire, it is necessary to protect against agents that may loop through a network or possibly corrupt data caches with expired data.

The *agenda* contains the agent's application-level goals, including information about the agent's intended task(s), the resources required to perform those tasks (e.g., file or network access), and the expected cost of performing the tasks.

(e.g., in terms of communication bandwidth or CPU time). An agent provides descriptions of its intended task and resource requirements, and these specifications are used as part of the authentication process between the agent and the platform. In addition, agenda information can aid the platform in determining the appropriate privileges to award an authenticated agent. To protect in transit information, the agent may encrypt portions or all of the agenda to ensure the secrecy of its tasks. In such cases, the target platforms for the agent must already be in possession of the proper key to use in decrypting the agent's agenda.

The *itinerary* submodule contains the agent's travel plan, designates the platforms the agent intends to visit, and also grants permissions to clone the agent. For each target host platform, the itinerary contains the host's unique identifier, the host's public key, material for authenticating the host, and, finally, a checksum of the software expected to be running on the host. The platform's unique identifier and public key are used by the agent to authenticate the platform, but other authentication materials, such as session tokens, may also be available to augment these more standard materials. The checksum is used by the agent to verify that the platform has not been modified from the expected execution environment, which could indicate a recent update or a compromise in the system. In addition to the above components, the itinerary is also used to designate whether an agent will permit platforms 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. Finally, the agent's true destinations can be obscured to hide an agent's association with a platform or prevent observers from understanding the purpose of the agent. This protection can be applied to individual entries of the itinerary without impacting an intended recipient's ability to receive the agent. The mechanisms behind this encryption are discussed in Section 2.2.

The final submodule in the agent's main module contains the agent's *credentials*, which define a *tamper detector* comprising a public key, a signature, and the agent's prescribed authentication methods, *authentication submodules*, which describe which algorithms an agent can use to authenticate with platforms. The platform must support at least one of an agent's prescribed methods, or the two will not be able to authenticate. Credentials are used to capture the identity of the agent, and the entries in this component allow the agent to be authenticated across domains with dissimilar authentication mechanisms.

The *creator key* shown in Fig. 1 is used to help protect all sub-modules within the main module. When the agent is completely assembled and prepared for dispatch, the creating entity will create an asymmetric key pair. The public key will be added to a list of keys used to sign the agent, and the private key will be used to sign the agent. In order to allow for agent cloning, the agent carries a *signing keys list*, and as the name suggests, the list will contain a list of all platforms that have cloned and signed an instance of the agent. This method of cloning is used for two reasons. First, the agent's main module must remain intact, so future platforms the agent reaches can validate the integrity of the

agent. The second reason is more of a trust issue. Since the itinerary contains platform identifiers and public keys as well as permission to clone the agent, future target platforms can easily verify the validity of a cloned agent. If an agent is cloned but the platform did not have the proper permissions, the agent is considered illegitimate and can be destroyed and reported once it is discovered.

In addition to its main module, an agent may contain a dynamic module, the lower rectangle in Fig. 1. This module stores vital state and process data with a high degree of confidence as the agent moves from platform to platform. Its explicit separation from the main module also protects the crucial information described above from modifications that can occur in the dynamic module. Within the dynamic module, 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. A *digest* provides a mechanism of verification for this data. Before departing a platform, the agent creates a hash of the execution state and application data (using a function like SHA-512) 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 the key does not match or the digest is wrong, the agent will self-destruct.

2.2 Implementing Secure Modular Agents

SMASH agents are designed to be resilient against many of the security attacks found in modern day systems yet remain flexible through platform interaction. Our framework builds on past work in agent systems, but integrates multi-directional security into the design from the ground up. SMASH is a multi-agent system built on top of the *Security Enhanced Linux* and uses Python as the execution environment for the agents.

General Agent Implementation. Agents are written in the Python scripting language to provide an easy-to-use interface to the application developer. Python is a powerful object-oriented language whose features make it attractive for rapid application development. In addition, the separation of the agent implementation (written in Python) from the platform implementation (described in the next section and written in C++) provides a layer of abstraction between the agent's security policies and the platform's implementation of those policies.

The Python interface for defining a SMASH agent 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. Each of these submodules *except the agent's code* (i.e., the agenda, the itinerary, the credentials, the TTL, and the application data) are represented by additional Python classes provided in the SMASH middleware implementation. When a SMASH agent is first created, its `__init__` method is invoked, and, within

this method, the submodule components are either received as parameters or created. When a SMASH agent arrives on a new platform, the Python interpreter uses boot-strapping methods within the agent to load essential environment variables and to prepare the agent for the platform's admission process. This entire process is described in more detail in Section 3.

One final aspect worth noting about this programming interface is the ease with which the developer can specify initialization of the submodules. For example, as described next, an agent's agenda is represented using an XML-like definition. To initialize the agenda submodule, 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 agenda's details are implemented within the middleware. The agent's itinerary (which includes various information about each of the agent's target platforms) is also defined via a standard XML format and can also be automatically processed. Similar standard approaches for representing the other submodules 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. When using the `strong_run` method, 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. The agent 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 into the remainder of the middleware to find the next platform in the itinerary and move there.

Defining Expressive Agent Agendas. An example agent agenda is depicted in Figure 2, which shows the goal definition for a network event monitoring agent. The agent collects *any* of the *high* severity events that occur on network sensors. After identifying an event, it is hashed by the destination port and event name, so similar events on various sensors can be correlated. During the correlation, the agent counts similar events, and if any of the counts surpass the threshold, then the agent will retain these events. In this case (with a threshold of one), the agent will carry all events that are identified from the past 24 hours.

In cases where the agent would like to protect the goals, tasks, or resources from observers, the agenda entries can be encrypted for particular platforms using either symmetric or asymmetric cryptography. While other methods can be incorporated into our framework, we have defined the Secure Agent Container Transport Method (SACTM). SACTM is a single-use cryptographic container that allows both the agent and platform to validate the contents. The container is embedded in the agent before the agent is deployed, and the container is created with a symmetric key created during a secure key agreement between the agent's creator and the target platform. The creator also creates a random

```
<GoalType = NetworkStatusReport>
  <Task>
    <NIDSQuery>
      <attribute> Description = "NIDS Event Query"</attribute>
      <type=HashedQuery>
        <attribute> EventType= "ANY,HIGH" </attribute>
        <attribute> HashBy = "DstPort,EventName" </attribute>
        <attribute> TimePeriod="Last Day" </attribute>
      </HashedQuery>
      <type=EventCorrelation>
        <attribute> GetCount  = "TRUE" </attribute>
        <attribute> TrackTime = "FALSE" </attribute>
        <attribute> KeepHostId= "FALSE" </attribute>
      </EventCorrelation>
      <type=EventFilter>
        <attribute> EventThreshold  = 1 </attribute>
        <attribute>  = "FALSE" </attribute>
        <attribute> KeepHostId= "FALSE" </attribute>
      </EventFilter>
    </NIDSQuery>
  </Task>
  <Resources>
    <Internal>
      <attribute> ProcessingTime = "300s" </attribute>
      <attribute> SensorDBAccess = "TRUE" </attribute>
    </Internal>
  </Resources>
</NetworkStatusReport>

<GoalType = SACTM>
  <attribute> PublicKey= AKey </attribute>
  <attribute> Nonce = 8686868 </attribute>
  <attribute> Data =   ...DATA... </attribute>
</SACTM>
```

Fig. 2. Model of an Network Event Monitor and Encrypted Goal

nonce and an asymmetric key pair, which are used to create a *seal* that is used by both the agent and platform to validate the SACTM. Essentially, the private key is used to sign the nonce and data, and the resulting seal is appended to the data and encrypted with the key. The agent creator then appends the public key and nonce to finish the SACTM, and after the container is created, the creator destroys the container key, leaving the only copy in the possession of the platform. The SACTM is verified after the agent and platform mutually authenticate. The platform will decrypt the SACTM data with the stored key and use the nonce, the public key, and the decrypted data to check the *seal*. If the check succeeds, the platform can ensure the SACTM retains its integrity. Next, the agent performs the same check. The novel feature of this container is that if either element tries to lie, the other will be able to detect the lie through the integrity check, so the platform cannot pass-off data not in the SACTM to the agent, and the platform will be able to detect a masquerading agent.

Finally, the agenda can also be used as a dossier or condition upon which the agent is admitted to the platform, and, if the agent violates the agenda constraints, the agent can be removed from platform.

An agent's itinerary is also implemented as XML-like specifications, and portions of it (e.g., single destinations) can also be partially secured in much the same manner. The details of these approaches are omitted for brevity.

3 Platform Components and Security Measures

Like SMASH agents, the host platform is engineered to provide support for an open architecture with high levels of security. This section describes the details of the platform that support the mobile SMASH agents, starting with a description of the model, including the flow of agents and information through the model, and concluding with a brief description of some implementation details.

3.1 The SMASH Platform Model

As shown in Fig. 3, we use a layered approach to compartmentalize our architecture and to prevent an outbreak of malicious activity. At the lowest level, 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. At this point the agent is considered to be an admitted *anonymous agent*. 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 authorization, the agent is placed into the Trusted Containment Zone (TCZ) and is considered an *authenticated agent*. 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. The remainder of this section describes these layers and an agent's movement through them in more detail.

SMASH's final components, shown to the right in Fig. 3, represent publicly accessible platform resources. The Blackboard is a memory-constrained FIFO queue 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 way that they may interact with each other. 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).

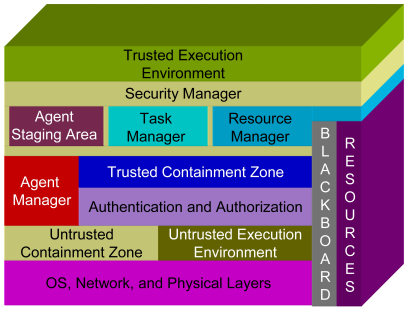

Fig. 3. SMASH Platform Architecture

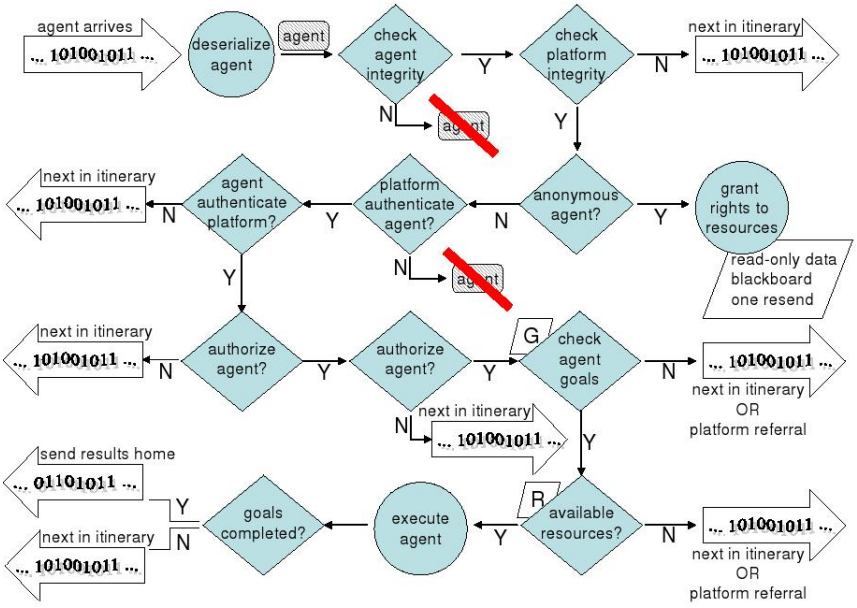

Fig. 4. Decision Tree Used by the SMASH Platform.

Fig. 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 to check the list of signatures and ensure the agent's integrity. The agent verifies the integrity of a platform by querying the TPM, which provides a signed hash of the platform's software, and the agent will compare this value with the one in its 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, may access the platform's lesser privileged services, or may 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, 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. 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 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, or even employ another mobile agent service [3] to identify and 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 the 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 identifies a task that is permissible and requires equal or lesser 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 initiates 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 agent execution, reduces idle time, and helps release resources in a timely manner.

When the necessary resources become available, the agent is moved into the Agent Staging Area (ASA), and its status is updated in the AM. In the ASA, the agent's Bootstrap Code (BC) is identified and loaded. 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. If any failure occurs, the BC aborts, and the agent self-destructs or returns home. Finally, the BC updates the agent's status within 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 restrict or kill the agent, or force the agent to leave. 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 a means to leave the platform.

3.2 Implementing a Secure Agent Platform

To enable several aspects of tamper detection in SMASH, our implementation utilizes Security Enhanced Linux (SE Linux [4]), which is a Linux kernel modified and partly maintained by the National Security Agency. SE Linux 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 [5]. In combination, this hardware and operating system enable our implementation of the security and trust mechanisms outlined above.

As described in Section 2.2, we use the Python programming language to provide a programming interface for defining agents. For the platform, we use C++, on top of which the Python agents run. C++ is more amenable to interaction with the SE Linux operating system services, has better performance, and makes many of the subtle aspects described above 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 Meeting a Wide-Range of Security Requirements

Multi-agent systems are very difficult to secure, simply because they invite foreign pieces of code to execute and fulfill a goal or objective. Even under the most ideal situations, security becomes highly complicated and requires applying not only cryptographic but also procedural measures to overcome threats and vulnerabilities in a system. A secure system typically satisfies properties of

accountability, authentication, availability, confidentiality, and integrity [6]. The system must be able to authenticate users of the system, and in doing so ensure the user is who they say they are, commonly using any of the following factors: "what you know," "what you have," and "who you are." The next issue is availability, which implies that required services and resources will be available at least when they are needed. Confidentiality refers to the fact that information must remain secret through out the security cycle, and no information is leaked by processes acting on the data. The final property is integrity, and this implies two elements, data integrity and source integrity. Data integrity means that any data being processed via a security system should not be modified by unauthorized users whether intentional or unintentional. The second element is Source Integrity, and this item implies that the source of the given data or message is untainted and represents their true identity.

The underlying goals of *accountability* are to disallow deniability both on the part of the agent and the part of the platform and to be able to reconstruct events. SMASH ensures accountability by requiring mutual authentication, tracking an agent's states and resources on a platform, and governing an agent's access throughout its stay on a platform. If the platform detects abnormal behavior (e.g., an agent operating outside of its stated goals or resource requirements), the platform can intervene. Accountability of platforms may not be as exact, in part because, as the agent moves from platform to platform, it becomes difficult to determine exactly by which platform an agent was modified. Our use of the digest and its sequential keys helps in this process, but the approach may still suffer from a risk of rogue platforms attempting to modify agents en route to other platforms.

Authentication is the process of identifying an entity and asserting with a high probability that this is the entity it claims to be and not an impersonator. In a dynamic environment, authentication is complicated due to the lack of persistent connectivity to a central authority. Common approaches to authentication require a central host or certificate authority to provide information about the identity bound to the key in question. Currently, SMASH relies on a model in which platforms and agents alike have *a priori* information about other entities that enable authentication. Such an approach incurs a good deal of initialization or setup costs that may be unreasonable in a mobile environment. Other approaches in dynamic environments handle this authentication requirement in a different manner, for example through quorum-based authentication [7]. Future work will investigate the feasibility of incorporating similar approaches into the SMASH architecture. To implement the actual authentication process, SMASH uses Pluggable Authentication Modules (PAM), which interface with the Pluggable Authentication Service, to perform the authentication on the platform.

Availability emphasizes how components and the system as a whole address incidental and intentional failures. Incidental failures may occur when an agent loses a network connection, and the agent does not handle the resulting exception created by the incident. An intentional failure is due to a malicious entity actively engaging the system in an attempt to disrupt or compromise services

and resources in the system, resulting in instability. SMASH focuses on ensuring stability from within and accomplishes this feat by applying a layered security approach. The first line of defense begins with the agent and its creator. In this layer, agents are coded in a defensive manner such that exceptions are caught and, to some extent, data and code are validated before being executed. The next line of defense falls within the platform. First of all, to prevent collateral damage, agents are executed in their own execution contexts using SE Linux [8]. Under this condition along with the *principle of least privilege* and SE Linux's access controls, agents are contained and unable to escalate their privileges, and once the platform detects the abnormal behavior, the agent is killed and system checks are performed to ensure everything is in order. If the platform becomes unstable, SE Linux is also used to contain this system, so it can actually be halted, reinitialized, and restarted into the last known good state.

In SMASH, *confidentiality* and *integrity* focus on keeping messages secret and intact. Confidentiality is typically accomplished through cryptographic measures, but methods like obfuscation can also be utilized to embed secret meanings into the existing messages, without changing the cover message. SMASH embraces current cryptographic techniques to accomplish secrecy, since these methods are proven secure and practical in real world environments, using algorithms like RSA, ECC, AES, etc. The SACTM is a slight exception. While it has not yet been proven secure, SACTM makes use of secure algorithms and protocols to help reinforce its security. Integrity is accomplished by using digest functions in conjunction with asymmetric cryptography. Digest functions are are non-invertible functions, meaning outputs can not be used to derive the inputs. This property allows information to be given a *probabilistically* unique value, where collisions (e.g., another input with the same output) are highly unlikely. To ensure information pertaining to the digest and the digest itself cannot be modified en route to a platform, asymmetric cryptography is applied to the digest, thus retaining the originality of the message.

5 Modeling Agent Interactions

The previous sections introduced SMASH's components, their respective functionalities, and their security guarantees. This section presents a real world example in which SMASH can be used to securely transmit mobile agents among platforms, providing an improved implementation of a common application. Specifically, we present the use of SMASH to support *epidemic updates* on, for example, commercial automobiles. As discussed in Section 1, epidemic updates can be used to intelligently propagate software updates to distributed platforms.

The implementation of this application begins on the factory floor, when the automobiles are originally manufactured. When the manufacturer creates a new automobile, it loads the vehicle with specific cryptographic keys, and the keys for the device are saved in the manufacturer's database as well. This initial "centralization" removes the need for a third party to be involved in verification processes at a later date. At some later time, the manufacturer may

identify a (non-critical) software update that it would like to distribute to certain automobiles. While a dealership is servicing one of these automobiles, the vehicle can be given a mobile agent (or set of mobile agents) that can clone itself and move through vehicles, supplying the necessary software update. In this process, the maintenance personnel at the dealership loads a *carrier agent* onto the vehicle under service. As their name indicates, carrier agents carry the software updates to platforms targeted by the update. Carrier agents are given a specified TTL after which the carrier agents will self-destruct. If a carrier agent reaches an automobile that requires the update but has not yet received it, the carrier agent loads the new software (when the car is parked), and sends a *verification agent* back to the manufacturer. When the TTL for the initial carrier agents has expired, the manufacturer sends traditional recall slips to all un-verified automobiles requiring the software update. When a vehicle supporting a carrier agent reaches an idle state (e.g., is parked in a parking lot), it attempts to clone itself and send its clone to nearby SMASH platforms. Upon arriving at a SMASH platform, if the vehicle supporting the platform is not of the type impacted by the recall, or the vehicle has already been updated, the agent self-destructs. If it is, the carrier agent deposits the update and sits on the platform, proceeding to pass the new software to un-updated vehicles.

The first carrier agent is composed of the following material. The agenda describes the type of update being applied and the intended firmware version to update. The itinerary contains a list of (the platforms of) all vehicles impacted by the recall. When an agent clones itself to send to a new platform, it decreases the itinerary by the platform(s) it has already visited. Rather than specifically identifying the other platforms by unique id (in this case, likely the Vehicle Identification Number, or VIN), a carrier agent could identify properties of the vehicles it needs to visit. Adding such expressiveness to an agent's itinerary is left for future work. The admission process for a carrier agent from the anonymous status to the authenticated status uses the pre-loaded manufacturer's keys, and appropriate counterparts are carried by the carrier agent. The *code* and *app data* for the carrier agent contain the code for uploading the update, the update itself, and diagnostic scripts to test and ensure that the update was correctly installed. To indicate that a platform has successfully been updated, the installation also causes a marker to be written to the platform's blackboard that indicates success. Upon arriving at a new platform, any carrier agent first checks this blackboard, and, if the marker is apparent, the carrier agent self-destructs.

After an attempt to update the platform is made, a *verification agent* is sent back to the closest dealership or manufacturer. This verification agent carries information like log files and the diagnostic test results back to the manufacturer for records keeping and assurance that the update was successful. The files and logs sent back to the sender are encrypted with the their public key, which is already loaded on the platform or embedded in the carrier agent. The verification agent's agenda describes the agent as a courier, but the more revealing details about the agent are protected with encryption. While the use of an agent in this case at first seems unreasonable, the use of an agent will enhance the probability

that the agent will reach its destination because the agent can travel in an ad-hoc and intelligent fashion. A message sent in a traditional manner may not reach its destination due to the network dynamics.

6 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 [9] 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. [10], 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 [11], state appraisals [12], and even agent execution traces [13]. 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 [14] 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 [2] supports multiple agent

implementation languages and also differentiates anonymous and authenticated agents. Aglets [16] are applet agents based on Javascript. They provide conditional access rights and moderate access control based on aglet "identity."

Ajanta [17] 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 [18]. 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 privileges are unnecessary [19]. 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.

7 Conclusion

In creating and implementing this SMASH concept framework, we have created a flexible and expressive approach to defining secure mobile agent systems. This process has also elucidated several research issues for future work within the scope of improving the SMASH framework. As described earlier, a replacement language for the XML-like specifications of agendas and itineraries would help agents more flexibly define their plans and travel schedules. In addition, we plan to revamp the models of agent interactions within SMASH platforms to under-

stand whether any relaxation of behaviors can be allowed without sacrificing the stringent security guarantees we have provided. The current restrictions placed on interactions among agents restricts the degree to which emergent behavior can be codified, possibly limiting the applicability of the current SMASH framework.

Another major undertaking is the formalization of the SMASH security guarantees and an evaluation of these guarantees against formalized security requirements. Section 4 provided an informal discussions of such issues, but a more rigorous evaluation will aid in arguing the system's robustness to common threats. Such a model will also help us assess the impact of future changes to the framework both in terms of expressiveness and security. Within this formalization, we will represent not only the secure architectural components but also the agents, their structure, and their interactions. This will help us more clearly explicate the manner in which we obfuscate agents' agendas and itineraries.

In this paper, we have defined SMASH, a mobile agent system with a unique combination of openness and security. SMASH affords agents confidence about the platforms with which they interact and platforms confidence about the agents they choose to support. In addition, SMASH makes it possible for an agent to move among platforms in a limited fashion without having to authenticate with platforms where the agent does not require access to privileged services. When an agent does authenticate with a platform, the two-directions of security help the platform ensure the agent is safe and helps the agent ensure that the platform is legitimate and that it can provide services required by the agent. As a final innovation, to support robust but simplified agent creation, SMASH agents are created using the Python scripting language. These agents are then supported by a middleware implemented in C++ and supported by a Trusted Platform Module (TPM) to provide the underlying stringent security guarantees.

In summary, multi-agent systems have the potential to improve current applications and open the door for new applications. Over the course of this paper, we have discussed how to improve the security in multi-agent systems, while allowing for an open architecture. SMASH is a new multi-agent system model that builds on past system innovations and incorporates new and existing security technologies. The paper discussed not only what SMASH can do, but it also showed that a multi-agent system can provide and implement an infrastructure based on information assurance. The paper also illustrated an application for *epidemic updates* build on the SMASH middleware. Overall, SMASH has the potential to improve the programmability of highly secure mobile agent systems.

Acknowledgments

The authors would like to thank the Center for Excellence in Distributed Global Environments for providing research facilities and the collaborative environment. This research was funded, in part, by the NSF, Grant # CNS-0620245. The views and conclusions herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

1. Roth, V.: Obstacles to the Adoption of Mobile Agents. In: Proc. of the IEEE Int'l. Conf. on Mobile Data Management. (2004) 296–297
2. Gray, R.S., Kotz, D., Cybenko, G., Rus, D.: D'Agents: Security in a Multiple-Language, Mobile-Agent System. In: Mobile Agents and Security, London, UK, Springer-Verlag (1998) 154–187
3. Seleznyov, A., Ahmed, M.O., Hailes, S.: Agent-based Middleware Architecture for Distributed Access Control. In: Proc. of the 22^{nd} Int'l. Multi-Conf. on Applied Informatics: Artificial Intelligence and Applications. (2004) 200–205
4. The National Security Agency: The SELinux Project. http://selinux.sourceforge.net/ (2005)
5. Trusted Computing Group: Trusted Computing Group Hompage. https://www.trustedcomputinggroup.org/home (2005)
6. Stallings, W.: Cryptography and Network Security: Principles and Practices. 4 edn. Prentice Hall, Englewood Cliffs, NJ, USA (2006)
7. V. Pathak and L. Iftode: Byzantine fault tolerant public key authentication in peer-to-peer systems. Computer Networks, Special issue on Management in Peer-to-Peer Systems: Trust, Reputation and Security **50**(4) (2006)
8. McCarty, B.: SELinux NSA's Open Source Security Enhanced Linux. 1 edn. OŔeilly Media, Inc., Sebastobol, CA, USA (2004)
9. Jochen, M., Marvel, L., Pollock, L.: A Framework for Tamper Detection Marking of Mobile Applications. In: Proc. of the 14^{th} Int'l. Symp. on Software Reliability Engineering. (2003) 143–152
10. Page, J., Zaslavsky, A., Indrawan, M.: Countering Security Vulnerabilities in Agent Execution Using a Self Executing Security Examination. Proc. of the 3^{rd} Int'l Joint Conf. on Autonomous Agents and Multiagent Systems (2004) 1486–1487
11. Hohl, F.: A Framework to Protect Mobile Agents by Using Reference States. Proc. of the 20^{th} IEEE Int'l. Conf. on Distributed Computing Systems (2000) 410–419
12. Farmer, W., Guttman, J., Swarup, V.: Security for Mobile Agents: Authentication and State Appraisal. In: Proc. of the 4^{th} European Symp. on Research in Computer Security, Springer-Verlag (1996) 118–130
13. Vigna, G.: Cryptographic Traces for Mobile Agents. In: Mobile Agents and Security. Volume 1419 of LNCS. Springer-Verlag (1998) 137–153
14. Cabri, G., Leonardi, L., Zambonelli, F.: MARS: A Programmable Coordination Architecture for Mobile Agents. IEEE Internet Computing **4**(4) (2000) 26–35
15. Suri, N., Bradshaw, J.M., Breedy, M.R., Groth, P.T., Hill, G.A., Jeffers, R., Mitrovich, T.S., Pouliot, B.R., Smith, D.S.: NOMADS: Toward a Strong and Safe Mobile Agent System. In: Proc. of the 4^{th} Int'l. Conf. on Autonomous Agents. (2000) 163–164
16. Karjoth, G., Lange, D.B., Oshima, M.: A Security Model for Aglets. IEEE Internet Computing **1**(4) (1997) 68–77
17. Karnik, N.M., Tripathi, A.R.: Security in the Ajanta mobile agent system. Software—Practice and Experience **31**(4) (2001) 301–329
18. Marques, P., Santos, N., Silva, L., Silva, J.G.: 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)
19. Sun Microsystems: The Java 2 Platform. http://java.sun.com/j2se (2006)