

# Agent Trustworthiness

Lora L. Kassab<sup>1</sup> and Jeffrey Voas<sup>2</sup>

<sup>1</sup> Naval Research Laboratory  
Center for High Assurance Computer Systems  
Washington, D.C. 20375, USA

kassab@itd.nrl.navy.mil

<sup>2</sup> Reliable Software Technologies  
21515 Ridgetop Circle, Suite 250  
Sterling, VA 20166, USA  
jmvoas@RSTcorp.com

## Abstract

Agent-based technology could revolutionize the manner by which distributed computation is performed. The fact that the information returned by an agent to the agent owner cannot be validated by the owner is impeding the widespread adoption of agent-based computing. Our paper addresses this concern by proposing a new type of software assertion to increase observability by providing agent owner's with agent state "snapshots." These snap-shots provide agent owners with: (1) a means to determine whether its agent's results are trustworthy, (2) information to debug a roving agent, (3) a greater ability to meet real-time constraints, and (4) a means to identify hosts systems that are resource-deficient, grant insufficient access rights, or tamper with agents. We present a methodology and tool for selecting and embedding protective assertions into agent code. We also discuss how the information from the assertions is automatically analyzed. Although our proposed assertions are not foolproof, they make it much harder for an agent to be tampered with in ways that are not detectable by the agent's owner. This knowledge is paramount for the utility of an agent-based system.

## 1 Introduction

Agent-based technology is transforming distributed computing from a scientific "number crunching" exercise to a more flexible computing arena. The explosive growth of the Internet as a medium for communication, business, and electronic commerce has fostered the growing interest in agent-based systems (ABS). An agent's migratory behavior provides the ability to utilize an unbounded set of sources for information and computing resources. The salient characteristic of ABS is the conservation of network bandwidth; once the agent migrates to a host system, all subsequent computation is performed there.

The flexibility of agent-based computing is not without penalty since the value-added by ABS is defeated if: (1) malicious or errant hosts attack agents, (2) malicious or errant agents attack hosts, or (3) erratic Internet behaviors or resource scarcity pose intolerable time delays.<sup>3</sup> These possibilities testify why wide-spread use of ABS is impeded for critical applications. This paper presents a scheme that addresses (1) and (3). This scheme can also be modified to address (2), but that will not be in the scope of this paper.

Even though host systems have a "trusted computing base" security advantage over agents, security vulnerabilities threaten each key player in this paradigm:

---

<sup>3</sup>The concern over protecting agents during migration is minimal as protection can be achieved using well-known cryptographic protocols to transfer an agent securely.

<b>Agent</b>	Agent's have no privacy; hosts can spy out code, data, or control flow. A host system can tamper with an agent's code, data, or control flow. An agent's code can be executed incorrectly. An agent can be terminated. A host can lie about system call results, its identity, or any requested data. A host can snoop or interfere with any agent communication.
<b>Agent Owner</b>	Agent owner's suffer from the effects of <i>all</i> of the agent vulnerabilities. Agent owner's also suffer from not knowing if an agent's results are correct.
<b>Host System</b>	Agents can infect host systems with a virus. Agents can exhibit unintended or undesirable consequences during execution. Agents can greedily consume resources and cause denial of service.

Figure 1 illustrates the core problem of ABS: an agent owner does not trust its agent (once dispatched), and agents and host systems<sup>4</sup> do not trust each other (mutual suspicion). This lack of trust raises three critical questions: (1) If an agent owner cannot trust an agent's results, then what is the purpose of using this computing paradigm? (2) If an agent cannot trust the host system, then why would the agent migrate to that host system (if this is even possible to control)? (3) If a host system cannot trust an agent, then why would the host system allow the agent to execute? Even in a closed, "trusted" environment, these questions are not easily answered, because there is no guarantee that players are (and will remain) benevolent and cooperative. This lack of trust may ultimately determine whether ABS will receive serious attention.

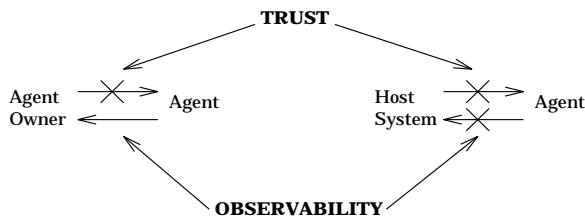


Figure 1: Trust Problems in an ABS

The many benefits from agent-based systems should not be overshadowed by these problems. There are approaches that can decrease the vulnerabilities of agents and agent owners. Our paper presents one such approach.

Our paper focuses on decreasing the vulnerabilities of the agent and of the agent owner to increase the reliability (and thus usability) of the agent computing paradigm. Others have addressed mechanisms for protecting host systems from the vulnerabilities of non-local code [8, 22, 30, 32]. Although protecting systems from mobile code is not a solved problem, sandboxing techniques and access control have been mostly successful for constraining non-local code. The problem of protecting agents from potentially malicious hosts has been deemed as the hardest security problem of this paradigm with very low solubility [10, 23]. This is primarily attributed to the fact that it is impossible to *prevent* malicious or faulty sites from tampering with cleartext agents [4] or from preventing *any* of the agent vulnerabilities listed above.

Recent research in mobile computing security contradicts the above statements by presenting a protocol that allows certain mobile code programs (ones that compute polynomial or rational

<sup>4</sup>We use the term *host system* as opposed to *agent server*, because the vulnerabilities of this paradigm affect the entire host system and not simply the agent server process executing on a machine.

functions) to execute in encrypted form except for the cleartext instructions [20, 21]. Therefore, execution on a host system does not compromise an agent’s privacy and it safeguards against agent tampering. Although this approach is critical for protecting mobile agents (which compute functions in this particular set of functions), we assume cleartext agents in order to encompass all agent computations.

Our paper presents novel ideas for fortifying agent-based systems with the ability to assess trust through *observability*, with a focus on increasing the observability of an agent’s state to its owner. The core of our technology is based on a new type of run-time software assertion, termed a *protective assertion*. We have borrowed the idea of using assertions to increase observability from the integrated-circuit design and software testing communities, who have known for years that increased observability provides increased trust. Protective assertions provide a dynamic strategy for an agent owner to observe its agents in order to make better decisions whether vulnerabilities have been exploited.

## 2 ABS Requirements

Before describing our approach, we will first identify a few necessary characteristics of a plausible security solution that do not limit the potential and flexibility of the mobile agent computing paradigm.

The most common approach for protecting agents and host systems has been simply to avoid migration to non-trusted hosts and not admit unknown code. This is commonly known as the “trust” approach. The problem with this approach is that it is absolute (*i.e.*, full trust or no trust) and it relies on *blind-faith* that all hosts and agents are consistently benign. Furthermore, this approach limits the intention that mobile agents exploit open and evolvable networks. That is, the network is “closed” and new hosts can only be integrated into the network if all other hosts trust the newcomers. Thus, we contend that any realistic, effective solution must presume a network of untrusted hosts.

In addition to assuming the presence of untrusted hosts, a plausible security solution targets detection instead of prevention. Techniques that make tampering more “difficult,” such as mutating agents in the time limited blackbox approach [10] is less viable than focusing on ways to *determine if* malicious host acts have been attempted. We contend that survivability of the agent owner is jeopardized if the returned information is incorrect or late returning. Therefore, mechanisms that *detect* either occurrence improve the *survivability* of the agent owner. We say this because if the agent owner is being deliberately fooled by incorrect information from its agents, the agent owner is almost certainly under some form of an information assault. If we can thwart such assaults, then the survivability of the agent owner is increased.

Our final requirement is the ability to track intermediate agent states. Mobile agents will be an important ingredient in producing secure, flexible distributed systems *if* there are advances in tracking the intermediate states created when mobile agents are executed [6]. Vigna has proposed one method for tracking intermediate states to detect the possibility of agent tampering through *traces*. These traces shadow the execution of an agent in a manner that cannot be forged by the host [23]. Tracking intermediate states, however, should not be equated with an interactive protocol. An interactive protocol between an owner and its mobile agents requires active participation from both an agent owner and an agent, precluding the possibility of the owner going off-line. Simply reporting information back to the agent owner provides the owner with the ability to better determine if the state of its agent is “sound” without forcing the owner to remain connected to the network (as these reports can be sent via email).

Notice the similarity between the trust problem associated with ABS and the problem associated with composing Commercial-Off-The-Shelf (COTS) software with customized software [27, 26, 25]. Developers who must rely on COTS functionality for certain components of their systems rarely trust it. It is a black box to them. Likewise, the state of a host system that a visiting agent will encounter is a black box to the agent owner. Further, an agent owner cannot know exactly how its agent will behave on a host system unless the owner has full access to the hardware and software of the host system. Because full access is not realistic, techniques that reveal the state of agents and the state of execution environments to the owners are warranted.

Throughout history, any time a process has been shrouded in secrecy, suspicions have arisen. For example, around twenty years ago, Joseph Newman attempted to get a patent for a perpetual motion machine from the US Patent and Trademark Office [15]. When asked by the office to reveal the internals of the machine, Newman refused, and no patent was granted. Today, the level of distrust of agent computation is similar to the disbelief that surrounded Newman’s invention. Had Newman let the patent examiners see inside and had the machine worked as claimed, the examiners would certainly have approved the application.

We contend that a security solution can be found that will provide a means for an owner to determine the legitimacy of its agents. Without state information about agents and hosting environments, the trustworthiness of an agent’s results can only be viewed with skepticism. Thus, to provide effective protection to agent owners without introducing large overhead costs or diminishing the flexibility of ABS, our approach will be based on following assumptions: (1) mobile agents execute in a network containing untrusted hosts, (2) detecting unexpected behavior by host systems is the key to protecting agents and agent owners, and (3) agent owners must track the states of their agents.

### 3 Protective Assertions

As stated earlier, our approach to increasing trust in ABS is based on the programming practice of using software assertions. Traditional run-time assertion checking enhances software validation by helping to ensure that program states satisfy certain semantic constraints. Some of the earliest documentation on assertions can be traced to [9, 14], and more recent research into giving programs the ability to check themselves during execution can be found in [18, 19, 3, 1]. While these ideas have generally addressed fault detection, our contribution to assertion theory is to modify it and apply it to increasing trust in ABS.

The conjecture that software assertions are invaluable as a complement to testing software is reinforced by others. Osterweil and Clarke classified assertions in their 1992 *IEEE Software* article as “among the most significant ideas by testing and analysis researchers” [16]. Microsoft reports copious use of assertions [13, 5]. In fact, interest in assertions has become so great that recent languages support assertions, including Anna [11] and Eiffel [12].

Software assertions are usually Boolean functions that evaluate to TRUE when a program state satisfies a semantic condition, and FALSE otherwise. Assertions can, however, output more than TRUE or FALSE, and in fact, can modify internal program states. Note that each assertion produces more program “output” than if the program did not have the assertions. This increases program observability.

Unlike the traditional use of assertions as a validation technique, our *protective assertions* increase the observability of a “seemingly black-box agent” to its owner.<sup>5</sup> This provides the detection

---

<sup>5</sup>The reason that we say “seemingly black-box agent” is that after a cleartext agent leaves “home,” there is no way to know whether the agent’s integrity is preserved.

that is necessary to add trust to ABS.

A protective assertion dynamically tests both the state of the agent and the state of the execution environment. Protective assertions can provide the following information to an agent owner.

- Protective assertions reveal any owner-specified agent state “snap-shots” throughout its execution at hosting systems. These “snap-shots” provide agent owners with a means to determine whether to trust its agent’s results (derived from one or more possibly untrusted hosts). Since agents are intended to be autonomous programs, protective assertions also provide an avenue to learn about decisions made by agents. Further, this information can also be used by a programmer to debug a roving agent.
- Protective assertions can “test” the execution environment. This not only provides agents with the insight as to when to migrate based on current resource consumption, access restrictions or resource expenses on the hosting system, but it also allows agent owners to better determine (and meet) real-time constraints for agent computations.<sup>6</sup> That is, agent owners can determine if and when other agents need to be dispatched in order to fulfill the overall agent computation.
- Finally, protective assertions provide the opportunity for owners to identify host systems that are malicious, resource-deficient, or do not grant sufficient access rights. This is valuable information for subsequent agent computations.

Thus, our protective assertions provide a window into the events that occur during an agent’s migration. This is currently not available, resulting in justifiably suspicious agent owners.

Our approach for providing trust is geared for Java agent-based systems, since Java is quickly becoming the development language of choice in a number of ABS: IBM Aglets, Mitsubishi Electric’s Concordia, General Magic’s Odyssey, and ObjectSpace’s Voyager to name a few. Therefore, there are several Java agent-based systems that could benefit from Java protective assertions. Even though our protective assertion tool is intended for Java agent-based systems, the theoretical underpinnings are independent of the agent-based system and capable of building this resilience into any agent-based system.

We assume that all agent owners and *host system owners* (this is the party that is responsible for launching the agent server process on that host system) own a public and secret key. Migrating agents are encrypted using the public key system PGP [17]. As in PGP, a random session key is generated for each agent. Using the private key algorithm, IDEA, the agent is encrypted with the random session key. Then, the RSA algorithm is used to encrypt the random session key with the recipient’s public key. Both the encrypted agent and the encrypted session key are dispatched from each agent migration point.

In order to transmit data from a host system back to the agent owner, the data is encrypted with the random session key that the agent carries. The agent owner can then use the random session key to decrypt the data received. Thus, cryptography is used to (1) securely transport agents and (2) return protective assertion results (as explained in Section 3.1). We assume that the keys for each hosting system have not been compromised.

### 3.1 Agent Protective Assertion Process

This section specifies: (1) the process by which protective assertions are embedded into agents, and (2) the process for how the results from the protective assertions are analyzed. These two

---

<sup>6</sup>We anticipate that use of resources will not be granted to visiting agents without a fee (this concept was employed in General Magic’s Telescript environment [31]).

processes are illustrated in Figure 2. The shaded blocks in Figure 2 represent “fortified” agents with protective assertions. Our complete approach consists of four steps in the following order: (1) employing fault injection [29], (2) using the Assertion Editor GUI, (3) instrumenting agents with protective assertions, and (4) generating an oracle. We will explain each of these in more detail:

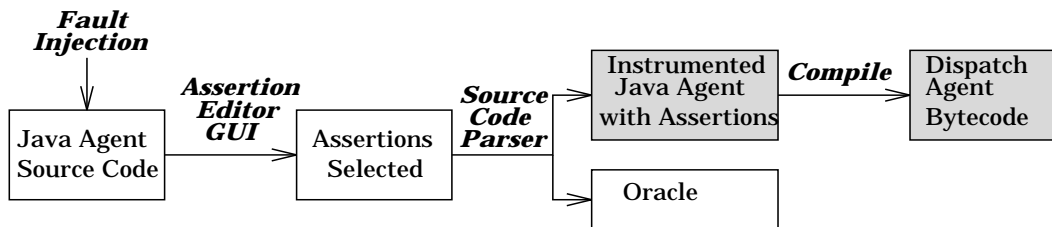


Figure 2: Process for Embedding Protective Assertions

1. **Employing Fault Injection:** Fault injection will identify agent weaknesses by simulating malicious, errant, or intolerable host system scenarios. As a result, fault injection will recommend what protective assertions are needed to “harden” the agent against weaknesses that fault injection isolates in an agent.
2. **Assertion Editor GUI:** Once the user has insight into which protective assertions are most necessary, the next step in our process involves the agent owner (or user) to (1) structure the protective assertions using the Assertion Editor GUI, (2) specify where the protective assertions are to be placed in the code, and (3) specify what information, if returned, indicates that the agent is no longer trustworthy.

Figure 3 is a screen-shot of the Assertion Editor GUI that is used to specify protective assertions. This GUI is part of Reliable Software Technologies AssertMate (TM) that is used to specify assertions for the Java language, and is also used to specify protective assertions for ABS.<sup>7</sup>

Note that, an agent owner can select protective assertions without first employing fault injection. In this situation, the selected protective assertions would depend on what information the owner deems sufficient for deciding whether an agent’s results are trustworthy. Once protective assertions are selected, our tool performs the remaining steps in the process without requiring additional user intervention.

3. **Instrumentation:** Next, our tool parses the agent’s Java source code and embeds the protective assertions into the agent.
4. **Oracle Generation:** In addition to parsing the code to embed protective assertions, our tool also builds an oracle. The tool creates an oracle daemon that executes on a designated machine to collect and analyze the information returned from executed protective assertions. The agent owner must provide the criteria for analyzing the results received by the oracle, *i.e.*, what return information from the agent is reasonable and what is not reasonable. Thus, a 1:1 mapping exists between the protective assertions and the information that the oracle is waiting to receive and analyze.

As an agent executes on a host system, the information generated from every executed protective assertions is collected locally (on a host system) and sent back to the oracle. If no

<sup>7</sup>Java does not support an assertion capability.

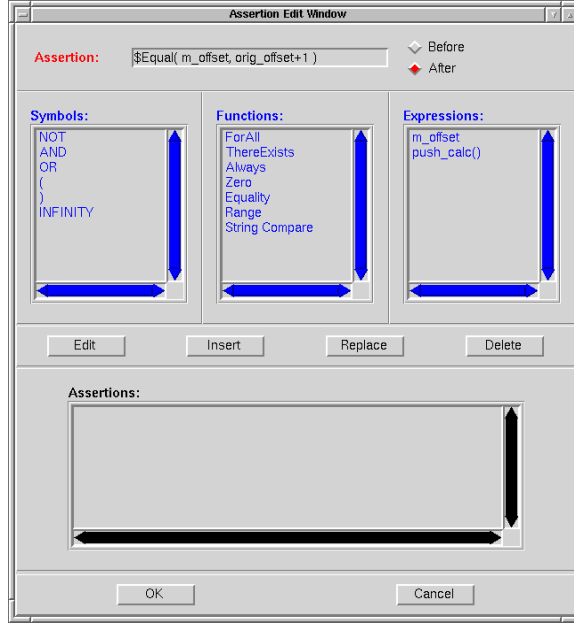


Figure 3: Assertion Edit Window

machine will be accessible throughout an agent’s migration, this information can be sent back via email. In this situation, once the agent owner is back on-line, the oracle can be executed using the email messages (one from each host system) as input.

Messages received from host systems include the outcome of each (uniquely identified) protective assertion that was executed. Messages are sent back to the agent owner encrypted with the random session key carried by the agent.

### 3.2 Agent Fault Injection

Since selecting the appropriate protective assertions is critical for producing “fortified” agents, we will explain in more detail how fault injection helps us make these decisions.

Fault injection is a family of techniques (that are similar to testing) which provide worst-case predictions for how badly a system will behave in the future. Though tools exist to automate fault injection, it is also possible (but more time consuming) to perform fault injection manually [24]. Regardless of whether fault injection is automated or manual, fault injection identifies potential agent weaknesses to an agent owner prior to agent dispatch.

Throughout an agent’s migration, it is possible for anomalous events such as: (1) a host system seeding an agent with misinformation and disinformation, (2) an agent exhibiting unintended or undesirable consequences during execution on a given host system, or (3) an agent reaching an unanticipated state during execution. Fortunately, fault injection can be modified to simulate each of these events. This allows agent owners to learn prior to dispatch how their agents will respond to such events in the “wild.” If agents are vulnerable to these events, then protective assertions are warranted.

Protective assertions harden agents by dynamically ensuring that the agent’s state remains “acceptable.” Fault injection provides insight into which agent computations need hardening. For example, a host system could tamper with state data or provide erroneous system call results. A host system could easily change values stored in memory, such that subsequent access causes an

agent program to access erroneous data. It is even easier for a host to advance the program counter to bypass a system call or not allow an exception to be thrown.

By simulating these types of events with fault injection, locations in an agent’s code can be identified that require hardening. Recognize that even if the embedded protective assertions are bypassed by the host system, this lack of information returning to the agent owner is a clear indication to the agent owner of probable malicious activity.<sup>8</sup> Our goal is not to attempt to prevent attacks by host systems, but rather to enable an owner to determine whether malicious activity is likely to have occurred.

Fault injection for Java agents is best applied to the input space and the host system interface. Since all arrays in Java are automatically bounds checked before they are accessed (and assuming there is not a bug in the JVM), buffer overflow is not a concern for Java programs. Further, Java programs cannot access random memory locations. Therefore, many of the standard ways that fault injection is applied to languages like C do not apply. However for other agent languages that do not enforce type safety or prohibit memory access, these standard fault injection anomalies can be applied to simulate such events.

Figure 4 illustrates how fault injection can be applied to Java agents. Here, we illustrate fault injection simulating malicious or errant host input or system call results.

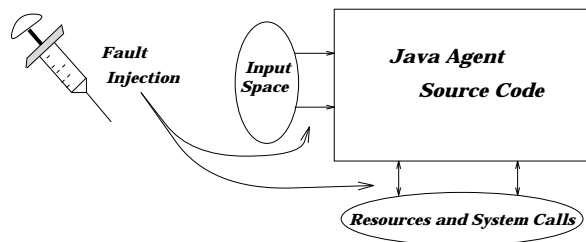


Figure 4: Agent Hardening using Fault Injection

Not only can fault injection aid in selecting protective assertions, it can be used to build recovery (fault tolerance) into agents. Although improving the fault tolerance of agents does not eliminate the possibility of malicious hosts killing agents, fault injection helps to identify situations where agent resilience can be improved. And fault injection can be re-applied after protective assertions are added to see if the added assertions have strengthened the weaknesses.

Note the utility of applying fault injection prior to dispatching an agent is highlighted by the ability to detect where appropriate checks are still needed. This improves overall agent performance by reducing the possibility that dispatched agents will terminate due to careless mistakes (e.g., an exception being thrown because an agent received unanticipated input from a host system).

Fault injection aids in selecting which protective assertions to embed and where to embed them. Metrics can also play a valuable role here: triggering the protective assertions to send back a message to the oracle daemon. Software metrics traditionally refer to measures of various structural entities of a program. In contrast, *semantic metrics* measure qualities such as reliability, the frequency of certain places in the code being exercised (these are termed as *profiling* metrics), execution traces through the code, *etc.* Protective assertions can utilize *profiling* metrics to measure the clock time spent executing certain parts of the agent by a host. For example, a system time probe assertion can be placed at the beginning and end of a code block.

This profiling metric serves several important purposes related to trust. If the time spent is

---

<sup>8</sup>A complete lack of information suggests that the agent was killed by a host system or the agent’s protective assertions were removed.

high (compared to the expected ranges expected by the oracle and thus determined by the owner), that tells the owner and the agent that the host is providing few resources to the visiting agent. If the time spent is unusually small or if no times are returned to the owner, then this suggests the host may have skipped over selected code or tampered with the agent. Regardless if the host tampered or skipped code, the agent owner now knows: (1) not to trust the agent's results, (2) to dispatch another agent, and (3) possibly which traversed host system was the culprit.

By combining fault injection with metrics, protective assertions can be crafted to make an agent more resilient to host system attacks and allow owners to inspect agent states more closely. This next section provides examples of types and format of protective assertions.

## 4 Classes of Protective Assertion

This section presents the classes of protective assertions that can be embedded into agent code. Assertions have the following format:

```
<assertion_type> ( <condition>, <message> );
```

<assertion\_type> is one of the following statements:

- The *ASSERT* statement performs a semantic check at a particular location in a program like the conventional use of assertions.
- The *PRE\_CONDITION* statement performs checks at method entry, the first statement in a method.
- The *POST\_CONDITION* statements perform checks at method exit, the last statement in a method but before any *return* statement.
- An *INVARIANT* statement checks whether the <condition> holds at the beginning and at the end of every method in a class.
- The *SYSTEM\_PROFILE* statement checks system conditions at program start-up to check whether a host system will provide a suitable execution environment.
- The *PROBE* statement is an assertion that unconditionally executes.

<condition> is any valid Boolean expression. An assertion fires when <condition> evaluates to the Boolean false. These expressions are the same as any legal control flow expression in the Java programming language. Our tool provides a means for simplifying the creation of complex Boolean expressions by allowing the use of existential quantifiers (*e.g.*, *ForAll*, *ThereExists*, *etc.*).

<message> is a text string that is displayed when an assertion fires. If our tool is used locally to “test” the code, then the <message> is displayed locally. Once an agent is dispatched, however, these <messages> will be collected locally on the host system and transmitted back to the agent owner when the agent completes execution on a particular host system.

*ASSERT*, *PRE\_CONDITION*, *POST\_CONDITION*, and *INVARIANT* are useful for capturing “snap-shots” of an agent's state, intermediate results, debugging information, and execution paths. Applying these assertion classes to agent code is not that dissimilar from the traditional use of assertions in non-mobile programs. For brevity, we will not elaborate on how these four assertion classes are implemented for Java source code, and will instead refer the reader to the NIST report [28] for detailed information. We will now focus our attention on the *SYSTEM\_PROFILE* and *PROBE* constructs. These assertions have been customized for Java mobile agent code.

The *SYSTEM\_PROFILE* assertion “tests” whether an execution environment is suitable. This statement can check for sufficient availability of system resources, or whether the hosting system will provide the necessary access permissions for an agent to complete its task. For example, the following assertion will fire if the amount of free memory is so insufficient that the agent’s computation will take too long to terminate. This situation could prevent the agent from meeting real-time constraints or it could cause the agent to exceed reasonable CPU resource expenses (assuming a time-based resource expense system is in place). If this assertion fires, the <message> is sent to the agent owner indicating that the agent is migrating.<sup>9</sup>

```
SYSTEM_PROFILE( Runtime.getRuntime().freeMemory() > TooLittleForMyAgent,
    ‘Agent Migrating: Insufficient memory ’ +
    Runtime.getRuntime().freeMemory() + ‘ at site:’ + getHost() );
```

Other *SYSTEM\_PROFILE* assertions can easily be created to check whether a host system grants the appropriate file permissions necessary for an agent to accomplish its task.

The ability to tamper with an agent’s code, data, or control flow information is considered one of the most grave security vulnerabilities of agents. Determining *when* tampering has occurred is a key challenge for our assertions and the oracle daemon. Protective assertions can capture state information such as values of scoped variables, intermediate results, information returned from system calls, database queries, whether code has been skipped, or any other dynamically requested information. Further, protective assertions can reveal the “constants” that should persist throughout migration. Inconsistency of this data is a clear indication of agent tampering. Once an agent owner knows that tampering has occurred, the question of whether to trust the results of an agent is immediately answered.

The *PROBE* assertion is best-suited for agent integrity checks.<sup>10</sup> Execution of this statement provides the ability to extract information about any class, including its methods, fields, and data. To provide this information, we employ Java’s core reflection API. The core reflection API supports introspection of the classes and objects within a JVM. We take advantage of this API within the *PROBE* assertion class as an opportunity to check the integrity of an agent’s code and data.

Our following example illustrates the *PROBE* assertion. Here, the agent owner will be notified of all methods in the class, *class\_name*. The <condition> parameter is set to false to force execution of this assertion.

```
PROBE( false, (Class.forName(class_name)).getMethods() );
```

The agent owner already knows exactly (assuming the code isn’t self-modifying) what method names should be returned. Likewise, similar reflection methods can be used to obtain underlying information about all constructors within a class, field values, exceptions thrown, modifiers, types, number of parameters, *etc.*

Since owners embed protective assertions prior to agent dispatch, the owner knows (via the oracle generated) what information is expected from the agent for each host visited. If there is a significant time lapse where no information is transmitted back to its owner via protection assertions, then this strongly suggests that: (1) the agent was terminated, (2) protective assertions were removed, or (3) other greedy resource consumptive processes have caused the agent to receive little CPU access (if any). This is true because the *PROBE* assertions must be executed. Regardless of which vulnerability was exploited, the benefit is that the agent owner detects this occurrence

---

<sup>9</sup>The symbol + denotes concatenation.

<sup>10</sup>The previous assertion classes can be used as well, however, our *PROBE* statement is unconditionally executed which is guaranteed to attempt to relay the information back to the agent owner.

and can react accordingly. Such knowledge is imperative for agent owners who must trust the information that they receive from their agents.

## 5 Summary

The growing dependence on information predicates the necessity for information trustworthiness. Unless the security weaknesses engendered by the ABS paradigm are adequately strengthened, agent-based computing will be shelved as an insecure technology searching for purpose. We have provided six classes of assertions that address these security weaknesses. Each assertion class boosts an agent's observability to its owner. We are currently investigating additional classes.

Protective assertions, like other ABS security approaches, are not foolproof. Even with protective assertions, it is still possible to clone agents, tamper with agents, remove assertions, lie to agents or tamper with agent communication. Although these malicious acts reduce the effectiveness of employing protective assertions, our technique provides the capability to make informed decisions *if* any malicious act was attempted. As stated earlier, if protective assertions are removed, the subsequent lack of information returning to an owner suggests foul play.

Our approach can be criticized for the overhead required to return information to the owner. We contend that this overhead is minimal relative to the computational bandwidth savings afforded by agents. A second criticism is that return data can be tampered with. However, only if the tampered data is consistent with what the oracle anticipates will it go undetected. And this seems unlikely unless host systems become omnipotent.

## References

- [1] J.M. BIEMAN AND H. YIN. Designing for software testability using automated oracles, In *Proceedings of International Test Conference*, pages 900-907, September 1992.
- [2] Boris BEIZER. *Software Testing Techniques*, Second Edition, International Thompson Computer Press, 1990.
- [3] M. BLUM. *Designing Programs To Check Their Work*, Technical report, University of California at Berkeley, December 1988.
- [4] David CHESS, Benjamin GROSOF, Colin HARRISON, David LEVINE, AND Colin PARRIS. Itinerant Agents for Mobile Computing, *IEEE Personal Communications Magazine*, 2(5):34-49, October 1995.
- [5] M. EVANS AND S. McCARRON. The Assertion Definition Language Translation System: An Automated Test Generation Tool, *Proceedings of Quality Week'94*, San Francisco, CA, May 1994.
- [6] William M. FARMER, Joshua D.. GUTTMAN, AND Vipin SWARUP. Security for Mobile Agents: Issues and Requirements, In *Proceedings of the 19th National Information Systems Security Conference*, pages 591-597, Baltimore, MD, October 1996.
- [7] A. GHOSH AND T. O'CONNOR. *Analyzing Programs for Vulnerability to Buffer Overrun Attacks*, Technical report, Reliable Software Technologies, January 1998.

- [8] Li GONG, Marianne MUELLER, Hemma PRAFULLCHANDRA AND, Roland SCHEMERS. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2, In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.
- [9] C. A. R. HOARE. An axiomatic basis for computer programming, *CACM*, October 1969.
- [10] Fritz HOHL. *Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts*, To appear in *Mobile Agents and Security Book* edited by Giovanni Vigna, published by Springer Verlag 1998.
- [11] D. LUCKHAM AND F. VON HENKE. An overview of ANNA, a specification language for Ada, *IEEE Software*, pages 9–22, March 1985.
- [12] B. MEYER. *Eiffel the Language*, Prentice-Hall, 1992.
- [13] B. A. MUELLER AND D. O. HOSHIZAKI. Using Semantic Assertion Technology to Test Application Software, *Proceedings of Quality Week'94*, San Francisco, CA, May 1994.
- [14] P. NAUR. Proof of algorithms by general snapshots, *BIT*, 6(4):310–316, 1966.
- [15] E. MARSHALL. Newman's Impossible Motor, *Science*, 10, pp. 571-572, February 6, 1984.
- [16] L. OSTERWEIL AND L. CLARKE. A Proposed Testing and Analysis Research Initiative, *IEEE Software*, pages 89–96, September 1992.
- [17] SIMSON GARFINKEL. *PGP: Pretty Good Privacy*, O'Reilly & Associates, Inc., 1995.
- [18] D. ROSENBLUM. Towards a method of programming with assertions. In *Proceedings of the 14th International Conference on Software Engineering*, pages 92–104, May 1992.
- [19] R. RUBINFELD. A mathematical theory of self-checking, self-testing, and self-correcting programs, Technical Report TR-90-054, International Computer Science Institute, October 1990.
- [20] Tomas SANDER and Christian F. TSCHUDIN. On Software Protection Via Function Hiding, Submitted to the 2nd International Workshop on Information Hiding.
- [21] Tomas SANDER and Christian F. TSCHUDIN. Towards Mobile Cryptography, *IEEE Symposium on Security and Privacy*, To appear in May 1998.
- [22] Joseph TARDO and Luis VALENTA. Mobile Agent Security and Telescript, In *Proceedings of IEEE COMPCON, February 1996*.
- [23] Giovanni VIGNA. Protecting Mobile Agents Through Tracing, In *Proceedings of the 3rd ECOOP Workshop on Mobile Object Systems, Jyväskylä, Finland, June 1997*.
- [24] J. VOAS. Fault Injection for the Masses, *IEEE Computer*, December 1997.
- [25] J. VOAS. A Defensive Approach to Certifying COTS Software, *IEEE Computer*, To appear in June 1998.
- [26] J. VOAS, F. CHARRON, AND K. MILLER, Robust Software Interfaces: Can COTS-based Systems be Trusted Without Them?, *Proceedings of 15th International Conference on Computer Safety, Reliability, and Security (SAFECOMP'96)*, Springer-Verlag, Vienna, Austria, October 1996.

- [27] J. VOAS, G. MCGRAW, A. GHOSH, AND K. MILLER. Glueing together Software Components: How good is your glue?, In *Proceedings of Pacific Northwest Software Quality Conference*, Portland, Oregon, Pages 338-349, October 1996.
- [28] J. VOAS, M. SCHMID, and M. SCHATZ. *A Testability-Based Assertion Placement Tool for Object-Oriented Software*, NIST Report GCR 98-735, National Institute of Standards and Technology, January 1998.
- [29] J. VOAS, G. MCGRAW, L. KASSAB, AND L. VOAS. A Crystal Ball for Software Liability, *IEEE Computer*, 30(6):29-36, June 1997.
- [30] Robert WAHBE, Steven LUCCO, Thomas E. ANDERSON, Susan L. GRAHAM. Efficient Software-Based Fault Isolation, In *Proceedings of the Symposium on Operating System Principles*, 1993.
- [31] James E. WHITE. *Telescript Technology: Mobile Agents*, General Magic White Paper, General Magic, Inc., 1996.
- [32] F. YELLIN. Low Level Security in Java, Technical Report, Sun Microsystems, 1995.