TOWARDS FAULT-TOLERANT MOBILE AGENTS

LORA L. KASSAB
Naval Research Laboratory
4555 Overlook Avenue SW, Code 5542
Washington D.C 20375
kassab@itd.nrl.navy.mil
Phone: (202) 404-4921
Fax: (202) 404-1017

JEFFREY VOAS
Reliable Software Technologies
21515 Ridgetop Circle, Suite 250
Sterling, VA 20166
jmvoas@RSTcorp.com
Phone: (703) 404-9293
Fax: (703) 404-9295

Abstract
The absence of a trusted computing base for mobile agents poses serious security issues for both the host system and the survivability of the agent. Once a mobile agent is dispatched, asserting anything about the host system, the agent's behavior, or even the agent's existence is difficult to ascertain. In order to employ agents with any degree of confidence, constraints need to be placed on the agent computation since no restraints can be imposed (or assumed) about the host system's hardware or software. This paper presents a fault-tolerant approach for increasing an agent owner's confidence in the integrity of its agent.

Keywords: Software Fault Tolerance, Mobile Agents, Agent Trustworthiness, and Voting.

1 INTRODUCTION

In traditional client-server settings, a central and trusted host communicates with statically bound client processes through either asynchronous messages or synchronous remote procedure calls. Mobile agents extend this communication paradigm by providing a more flexible approach for building distributed applications in an Internet-scale setting. A mobile agent is a program that is dispatched from a source computer and autonomously migrates to multiple hosts to perform the tasks (typically resource intensive computations) for which it was programmed.

The explosive growth of the Internet as a medium for communication, business, and electronic commerce has fostered the growing interest in agent-based systems. An agent's

migratory behavior provides the ability to utilize an unbounded set of sources for information and computing resources. The salient characteristic of agent-based systems is the conservation of network bandwidth; once the agent migrates to a host system, all subsequent computation is performed there. This approach is more efficient than moving the data to the computation and the overhead incurred by invoking remote operations.

Mobile agent computations can roughly be categorized into two domains: *independent* and *dependent*. The independent domain consists of mobile agents that are tasked with gathering data, usually from large databases. Computations are classified as independent when the result obtained from a host system does not depend on information retrieved from other hosts. This domain is typically associated with computations where the host systems are competitors, such as querying various airlines for flight information. The dependent domain includes computations where prior results obtained from host system(s) is required for subsequent computations at other host systems. Other research efforts include a third category commonly referred to as push technology, where agents provide host systems with information. An example is an agent that provides software upgrades to systems on the Internet. We will disregard the third category in this paper, because the permissions granted to the agents to accomplish such tasks pose an intolerable security threat to the host systems. Though there are other types of agent-based computations, the first two categories represent the typical strategies for employing agents.

The flexibility of agent-based computing is not without penalty since the value-added by employing agents 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.[1] Others have addressed mechanisms for protecting host systems from the vulnerabilities of non-local code [6, 17, 19, 22]. 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. Of the three problems, the third problem is deemed as the hardest security problem with low solubility [7]. This problem is considered most difficult primarily because it is impossible to *prevent* malicious or faulty sites from tampering with cleartext agents [2], nor is it possible for a cleartext agent to maintain code or data privacy.

Recent research in mobile computing security contradicts the previous statement by presenting a protocol that allows certain mobile code programs (ones that compute polynomial or rational functions) to execute in encrypted form except for the cleartext instructions [15]. Therefore, execution on a host system does not compromise an agent's privacy and it safeguards against agent tampering (because host systems cannot decrypt the agents). Although this approach is critical for protecting mobile agents (tasked with computations in this particular set of functions), we assume cleartext agents in order to encompass all agent computations.

Our focus is on the first problem: malicious or faulty hosts. This paper presents a method for decreasing the vulnerability of cleartext agents to malicious or faulty host systems. In addition to the malicious host system threats, agents systems are burdened with the problem that the behavior of an agent on each host system is unknown. That is, an agent will execute in many environments that may have not been anticipated or

[1] The concern over protecting agents during migration is minimal as protection can be achieved using well-known cryptographic protocols to transfer an agent securely.

tested before dispatching the agent. After all, the host system may simply suffer from faulty software or hardware, and we cannot anticipate what problems an agent may suffer from due to these non-malicious problems. In spite of these problems, the efficiency and flexibility of mobile agents has lured many to use this paradigm. But before the agent paradigm is applied to critical applications, assurance regarding the inherent security issues of mobile agents is needed.

This paper proposes a fault-tolerant approach that: (1) makes mobile agents more resilient to the possibility that host systems may attempt to tamper with agents, and (2) masks the effects of agent incompatibilities with host systems. Our approach assumes that we have complete control of the agent before it leaves our site, but after that, we have no control of the agent. We have designed an approach by which an agent can move from host to host and perform the tasks that any other agent can perform, while increasing our confidence in the integrity of the results from our agent. We expect that an agent will at some point accidentally visit a malicious or faulty host. Our approach presumes this likelihood and provides defensive steps that can attempt to overcome any damage.

2 AGENT INSECURITIES

As a simple example of the types of concerns we are worried about, suppose an agent is dispatched from an agent owner, O, to multiple host systems. After the agent leaves, how does O know its agent is alive "in the *desired* way"? Certainly, if O receives a simple "i am alive" message from the agent, then the alive portion of the question can be addressed. But assuring O that the agent is alive in the desired manner is not easily answered. To answer this, we need to know whether the agent is performing the computation x that it is supposed to (as opposed to performing some other computation y that it should not be). If the agent is performing computation y, that suggests that the host system may have maliciously tampered with the agent's behavior. However, performing computation y also suggests another possibility: a host has inadvertently tampered with the agent due to some unexpected incompatibility, such as a glitch in the host's software or hardware system. Either situation will likely force the agent to exhibit unintended behaviors, thus increasing the possibility that the agent will return incorrect results to the owner. From the standpoint of O, both situations are undesirable and equally difficult to detect.

If an agent were to return incorrect information, we must first be able to detect that the information is incorrect. Secondly, it would be beneficial to be able to determine which situation caused the problem: a faulty host or malicious host. Since determining this will often be too costly or impossible, the best that we can do is to build enough robustness into how agents return information to the agent owner such that the possibility of detecting either occurrence is increased.

One idea that appears to partially address the faulty or malicious host problem has already been proposed. Vigna has proposed cryptographic traces of the computations performed by agents [18]. These traces shadow the execution of an agent in a manner that cannot be forged by the host. Although, this information provides insight into an agent's status, manual verification of the "huge" ("even if compressed" [18]) statement traces is an expensive process, which is likely to be inefficient and error-prone. This conjecture is reinforced by others [1]: "The trouble with traces is that they give us far

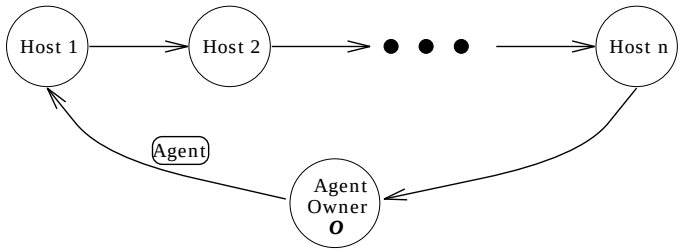

Figure 1: **A simple agent computation.**

more information than we need. In fact, the typical trace program provides so much information that confirming the path from its massive output dump is more work than simulating the computer by hand to confirm the path."

An alternative solution to increase the likelihood that agents are executing in the desired way is to build fault-tolerant schemes into agent-based systems. We are not the first to propose using the principles of fault-tolerant computing with mobile agents and will summarize some of the ideas put forth by others. But before doing so, we will first give an example of what it means to build fault tolerance into an agent-based system.

Figure 1 depicts a simple agent computation, where an agent is dispatched from the agent owner O, migrates to n networked host computers and returns back to O. For simplicity, we will assume that the source and final destination of the agent are the same. If a mobile agent is corrupted while traveling from host to host, the corruption will almost certainly propagate to each successor site including the final destination site. Thus this issue can be modeled as a "propagation problem," and there is a wealth of existing solutions from other domains that allow us to study how corrupted information propagates (e.g., static fault tree analysis, failure mode effect and criticality analysis [11], slicing [20], data flow analysis [14], etc.). We will leverage the basic principles of halting the propagation of corrupt information in agent-based systems by proposing a "design-for-fault-tolerance" approach to thwart the effects of agent corruption.

The IEEE defines *fault tolerance* as [8]:

> "(1) the ability of a system or component to continue normal operation despite the presence of hardware or software fault. (2) the number of faults a system or component can withstand before normal operation is impaired. (3) pertaining to the study of errors, faults, and failures, and methods for enabling systems to continue normal operation in the presence of fault."

Traditionally, fault tolerance has been achieved by building subsystems from redundant components that are placed in parallel to ensure higher quality [13]. Due to the improvement in hardware fault tolerance, the use of redundant components was extended into the software domain. For example, in an n-version system, two or more *independently created* yet functionally equivalent programs are developed from the same specification. A voter collects the outputs generated from executing the n versions in parallel, and selects the output of the fault-tolerant system according to a pre-specified algorithm. The general intuition is that the output value that is most frequently observed must be the correct result; this is of course not always true [10].

Schneider [16] presents a fault-tolerant approach to mask the effects of faulty processors by replicating the agent at each visited site. More specifically, the agent visits multiple host systems at a site and each host system sends the agent's output to every host system in the subsequent site where voting will occur to mask the effects of faulty processors. This approach also uses digital signature traces to allow the voter to verify the authenticity of its electorate.

Although this approach is fairly successful in coupling fault tolerance and agents, there are a couple drawbacks. If any processor involved in the fault-tolerant agent computation is down or refuses to allow the agent to execute, this fault-tolerant agent computation can easily degrade to the agent computation illustrated in Figure 1 unbeknownst to the agent owner. To prevent this from occurring, voters could be forced to delay voting until each member of the electorate submitted its results. In this situation, however, a processor that is either down or refuses the agent would cause the entire agent computation to never terminate. Either situation is equally undesirable since the goal is to build fault tolerance into agents. A second drawback is that whenever this fault-tolerant scheme successfully masks a faulty processor, the agent owner has no record of which host was malicious or faulty. This information is quite useful for subsequent agent computations.

Another fault-tolerant approach has been proposed which dispatches two identical agents to a known set of host systems, where one traverses the systems in one direction and the other agent does the reverse [21]. So for example if you order the hosts, $h_1, h_2, \ldots, h_n$, then one agent would traverse forward from h_1 to h_n, and the other agent would start at h_n and work backward. The author of this approach acknowledged that this approach is only successful in detecting at most *one* malicious system. Furthermore, this approach cannot determine which host in $\{h_1, h_2, \ldots, h_n\}$ is malicious to eliminate that host site for subsequent agent computations.

Some might also suggest a similar approach where multiple identical agents are released and try all sorts of host traversal combinations in order to increase the probability that at least one non-tampered agent will survive. We can conceive situations where this "shotgun" approach would work, but we also can see it flooding the Internet to the point of gridlock.

Our solution is based on the principles of fault tolerance but without any independence assumptions. Our protocol requires a limited degree of interaction between the agent owner (or another designated, trusted machine that will remain on-line) and its agent.[2] 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. This is the trade-off that we incur in order to increase the fault tolerance of agent-based systems. The interaction involved, however, must not be equated with agent task-delegation. Further, the communication overhead does not even come close to negating the advantage of employing agents to conserve network bandwidth to perform a typically resource intensive task with local resources/data on a host system. Rather, this interaction is merely to allow the voting portion of our fault-tolerant scheme to execute on a trusted system, which for more critical agent applications is necessary.

Similar to the fears of using Commercial-Off-The-Shelf (COTS) components, knowing

[2]For the remainder of the paper, we will assume that agent owner communicates with its agent even though it is just as feasible to designate another "trusted" machine.

a priori how an agent will congeal with a given host would require omnipotence. With agents, the state of the host systems that they will traverse is a black box to the agent owner. It is not possible to ever know exactly how an agent (the software) will behave on a host system without full access to the hardware and software of the host system. Therefore, applying defensive fault-tolerant strategies to agents with a limited degree of interaction with agent owners is all that we have to work with, since we cannot alter the behavior of the hosts (unless we wish to develop malicious agents, which of course we do not).

3 TOWARDS AGENT FAULT TOLERANCE

In our approach, we assume that all agent owners and *host system owners* (these are likely to be the principals responsible for launching the agent server process on the host system) own a public and secret key. Further, we assume that migrating agents are encrypted using a public key system similar to PGP [4]. 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 migration point of the agent.

In order to transmit data from a host system back to an agent owner, the data is encrypted using the random session key that the agent carries. Next, the random session key is encrypted (by the host system) with the host system's secret key. The agent owner can then decrypt the encrypted random session key with the host system's public key, and then use the random session key to decrypt the data received. Thus, combining PGP with agents allows us to: (1) securely transport agents, and (2) return computational results from each host system. This approach assumes that the keys for each host system and the agent owner have not been compromised.

By adding fault tolerance to agent systems, the problems created by malicious systems can be decreased and assurance that the agent is compatible with a host system can be increased. This is quite different from the simple "fire-and-forget" methodology.

Similar to the aforementioned IEEE definition for fault tolerance, *software fault tolerance* refers to the ability of the software to produce "acceptable" outputs (as defined in the system-wide requirements) regardless of the program states that are encountered during execution. In the context of agents, our goal is to mask agent/host incompatibilities and mask errors resulting from defective or malicious hosts. To satisfy this goal, we will employ the ideas of *redundancy* and *voting*.

We will now discuss our scheme for applying *redundancy* and *voting* to improve the validity of the results from agents. Our approach deliberately dispatches an agent to visit multiple host systems within a single domain whenever enough information exists to do so.[3] Doing this improves the chance that the results returned from the agent will be correct. Naturally, we cannot force different domains to have multiple host systems. But if these domains have redundant host systems, then the agent will visit multiple host

[3]It is not always the case that a single domain will have redundant host systems, however in this paper, we will take advantage of that situation whenever possible.

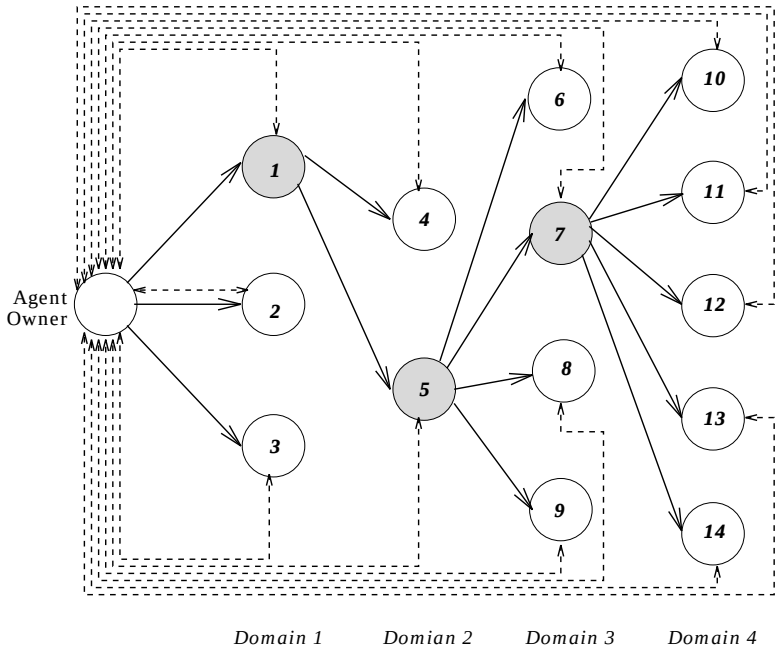

Figure 2: **Fault-Tolerant Agent Architecture.**

systems. As you will see, this (combined with voting) boosts the fault tolerance of mobile agents.

Figure 2 illustrates our redundant agent architecture. Each column in the figure corresponds to a domain containing multiple host systems that are visited by the agent. Here, O is the agent owner, and *Domain 1*, *Domain 2*, *Domain 3*, and *Domain 4* are the respective domains that are traversed by the agent. In this figure, our agent will traverse four different domains, for a total of 14 host machines. The solid lines in the figure designate the path of the agent, and the bidirectional dashed lines represent the transmission of the encrypted results computed from executing the agent on one of host platforms and an agent owner response.

We will now walk through this example of a fault-tolerant agent system. It is the role of the agent owner to dispatch the encrypted agent and random session key (as we previously described) to the three host systems in *Domain 1*, namely 1, 2 and 3. Each of these host systems decrypt the random session key using its own (the host system's) secret key. With the random session key, each host system can now decrypt and execute the agent. Upon completion, each host system sends the encrypted results (not the agent) and random session key back to the agent owner. For now, let's assume every host system returns these results punctually.

After the results are obtained from each of the host systems within a domain, the agent owner votes on the outputs received and selects an output for this domain based on a pre-specified voting algorithm. Then, the agent owner generates a new random session key and sends the key to the host system that had the "correct" output. If more than one host system provides the "correct" output to the agent owner, then one of these host systems can be selected at random. In *Domain 1*, host system *1* had the "correct" output and therefore received a new random session key. In Figure 2, note that the "selected"

hosts correspond to the shaded host circles. Next, the selected host system dispatches the agent with the new random session key to each of the host systems in the succeeding domain.

In Figure 2, the unshaded host systems in each domain are notified by the agent owner to terminate the agent. It is possible that the agent does not get terminated, and in fact it might get dispatched to additional other host systems. However, if any host system attempts to return information back to the agent owner, the owner will detect that the wrong random session key (an out-dated key) is in-use and will discard any results that are not encrypted with the correct random session key.

Because voting takes place on a trusted system, assurance in the output for each domain is increased.[4] By passing intermediate results back to the agent owner, the owner now has insight into which host systems could be faulty or malicious and can dispatch future agent computations accordingly.

This idea of voting on a trusted base has been championed by others. Farmer *et al.* [3] stated that as one achievable security requirement of mobile agents, any critical decisions should be made on a trusted system. Since correct execution of instructions cannot be guaranteed [5], voting on an a untrusted host system is not prudent. Further, voting is typically a simple computation. Thus, we contend that the voting algorithm is likely to be the easiest and most appealing portion of the agent to tamper with.

Different voting algorithms exist [9]. Typically, voting algorithms pick the output value that is most commonly observed. This is referred to as "majority voting." If there is not a majority agreement, one output may be selected at random, maximum agreement may be chosen, or a median result may be computed based on the outputs received [12]. Using a majority voting scheme, unless a majority of the host systems within a domain collude to blatantly lie or tamper with the visiting agent in exactly the same way, this approach can mask the effects of up to $\lceil n/2 \rceil - 1$ faulty host systems where n is the number of host systems in a domain. For agents, the voting algorithm employed may depend, in part, on the type of information that the agent has collected, the agent's computation, and the number of host systems within a domain.[5]

As an example of how voting depends on the agent's task and current situation, suppose an agent can only visit two host systems within a domain. Further suppose the agent is tasked with finding the cost of a flight from point A to point B, and the result of the second system is different than the first. Since there is a discrepancy in this type of agent task, the cheaper price may be favored. Such discrepancies can occur since database query results may change frequently. However, if the agent's task was purely computational with identical inputs on both host systems and a discrepancy resulted, selecting an output may not be as trivial. In this situation, this may indicate that the agent should repeat the computation to settle on one result, or that the agent should migrate to host systems with equivalent resource availability in another domain to repeat the computation. How to handle these and other similar discrepancies may differ for

[4] We can ignore the issue of the "correctness" of the voting code on the owner, because it is no less likely this voting will be correct than if the agent carried the voting code (as does the fault-tolerant scheme in [16]).

[5] Note that when a domain only has a single host or a domain does not allow us to visit more than one host, voting using a single result in not necessary.

each agent. Determining how to handle these situations aids in selecting which voting algorithm is best suited for the agent computation.

As we alluded to in the previous example, this voting scheme can be used for shopping agents. Certainly, dispatching multiple agents to make the same purchase is not the intent. Rather, voting can be employed to notify which agent should make the purchase. It is possible that each agent will get a different price from a host system even if the hosts operate properly. This is because the agents may execute at slightly different times/rates, thus getting different inputs and results as the world changes.

In summary, we have achieved a higher degree of agent fault tolerance by programming the agent to visit multiple host systems within the same domain. The rationale for doing so follows: different files, environment variables, permissions, and software and hardware configurations can cause each host system to be unique even when they are in the same domain. Software is a dynamic, behavioral entity, whose behavior is partially determined by the software's inputs as well as the immediate state of the execution environment. Since the very nature of agents is to traverse multiple systems (which means traversing multiple environments), the exact behavior of an agent in each different environment cannot be known until the exact time at which the agent is in each environment.

By tracking intermediate results, an owner can better determine the legitimacy of its agents for both dependent and independent computations. This also allows agent owners a means to better determine (and meet) real time constraints for agent computations. That is, if an agent is tied up due to resource deficient host systems or if an agent was terminated at some point, an agent owner can learn (or deduce) this by receiving (or lack of) these intermediate results and dispatch other agents accordingly. Otherwise, the agent owner has little, if any, information about the status of its agents.

4 SUMMARY

Admittedly, there are a couple drawbacks to our approach. First, the agent owner or another trusted system must remain connected to the network while the agent is still active since communication is continual during the agent's migration. Thus, this approach would not be useful for those who need to remain off-line and have no access to other machines that will remain on-line. Second, the communication overhead to send information back to the agent owner will delay the agent's computation. This will increase the duration of the agent's overall computation, especially when the voter is waiting to receive results from other systems.

As we previously mentioned, however, the voter (agent owner) need not wait for all of the results. For instance, with a majority voting scheme, a voter need only wait until it has received a majority of identical results. Thus, voters can tolerate slowdowns in returning results or even situations where agents have been terminated by a host system. In spite of these drawbacks, we remain firm in our belief that when integrity is *imperative*, it is better to incur the additional costs of returning information to the agent owner and letting it perform the voting than risking voter corruption by host systems. Furthermore, arming an agent owner with intermediate results is critical in determining the trustworthiness of an agent's final results.

This paper has addressed the problem of agents visiting anomalous host systems.

Employing this methodology for agent-based systems provides increased assurance that agents have not been accidentally or maliciously tampered with. This methodology utilizes a combination of the principles of fault-tolerant computing (redundancy and voting) and cryptography.

We have recommended that the following guidelines be adopted to increase agent assurance:

1. Generate a new random session key before dispatching an agent from *every* migration point,
2. Visit multiple host systems in a domain whenever possible,
3. Return the encrypted intermediate results from each host system to the owner, and
4. Employ voting at the owner.

Note that our scheme does not limit the user from the many benefits of agent technology. Instead, we have changed the manner by which the agents are dispatched, distributed, and compute results to reduce the concern about malicious or faulty hosts and thus increase the confidence an agent owner can place in its agents results.

Acknowledgements

Jeffrey Voas has been partially supported by DARPA Contracts F30602-95-C-0282 and F30602-97-C-0322, National Institute of Standards and Technology Advanced Technology Program Cooperative Agreement Number 70NANB5H1160, and Rome Laboratories under US Air Force Contract F30602-97-C-0117. THE OPINIONS AND VIEWPOINTS PRESENTED ARE THE AUTHOR'S PERSONAL ONES AND THEY DO NOT NECESSARILY REFLECT THOSE OF THESE AGENCIES.

References

[1] Boris Beizer: *Software Testing Techniques*, Second Edition, International Thompson Computer Press, 1990.

[2] 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.

[3] 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.

[4] Simson Garfinkel: *PGP: Pretty Good Privacy*, O'Reilly & Associates, Inc., 1995.

[5] Li Gong: *"Survivable Mobile Code Is Hard to Build"*, February 1997.

[6] 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.

[7] 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.

[8] IEEE: *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 729-1983, 1983

[9] B. W. Johnson: *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley, pages 54-59, 1989.

[10] J. Knight and N.G. Leveson: *"An Experimental Evaluation of the Assumption of Independence in Multiversion Programming"*, IEEE Transactions on Software Engineering, SE-12(1):96– 109, January 1986.

[11] D. J. Lawson: *Failure Mode, Effect, and Criticality Analysis*, Electronic Systems Effectiveness and Life Cycle Costing. Editor J. K. Skwirzynski, NATO ASI Series, F3, Springer-Verlag, Heidelberg, pages 55-74, 1983.

[12] Michael R. Lyu: *Handbook of Software Reliability Engineering* IEEE Computer Society Press, McGraw Hill, 1996.

[13] J. J. Marciniak: *Encyclopedia of Software Engineering*, Wiley, 1994.

[14] S. Rapps and E. J. Weyuker: *"Selecting Software Test Data Using Data Flow Information"*, IEEE Transactions on Software Engineering, 11(4):367-375, April, 1985.

[15] Tomas Sander and Christian F. Tschudin: *"Towards Mobile Cryptography"*, IEEE Symposium on Security and Privacy, pages 215-224, May 1998.

[16] Fred B. Schneider: *"Towards Fault-tolerant and Secure Agentry"*, Invited paper, 11th International Workshop on Distributed Algorithms, Saarbrücken, Germany, September 1997.

[17] Joseph Tardo and Luis Valenta: *"Mobile Agent Security and Telescript"*, In Proceedings of IEEE COMPCON, February 1996.

[18] Giovanni Vigna: *"Protecting Mobile Agents Through Tracing"*, In Proceedings of the 3rd ECOOP Workshop on Mobile Object Systems, Jyvälskylä, Finland, June 1997.

[19] Robert Wahbe, Steve Lucco, T. E. Anderson and Susan L. Graham: *"Efficient Software-Based Fault Isolation"*, In Proceedings of the ACM SIGCOMM 96 Symposium, ACM, 1996.

[20] M. Weiser: *"Programmers Use Slices when Debugging"*, CACM. 25(7):446-452, July, 1982.

[21] Bennet S. Yee: *"A Sanctuary for Mobile Agents"*, DARPA Workshop on Foundations for Secure Mobile Code, February 1997.

[22] F. Yellin: *"Low Level Security in Java"*, Technical Report, Sun Microsystems, 1995.