# Towards Fault-tolerant and Secure Agentry*

Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853
USA

**Abstract.** Processes that roam a network—agents—present new technical challenges. Two are discussed here. The first problem, which arises in connection with implementing fault-tolerant agents, concerns how a voter authenticates the agents comprising its electorate. The second is to characterize security policies that are enforceable as well as approaches for enforcing those policies.

## 1 Why Agents?

Concurrent programs are traditionally described in terms of *processes*, an abstraction that hides the identity of the processor doing the actual execution. This hiding allows processes to be implemented by time-multiplexing one or more identical processors, making the process abstraction well suited for concurrent programs—like operating systems and scientific codes—that will be executed on uniprocessors and multiprocessors.

It is unreasonable to suppose that the processors comprising even a modest-sized network would be identical. Moreover, a user's security authorizations are likely to differ from site to site in a network, so even were the network's processors identical, they would not necessarily behave in that way. An abstraction that hides processor identities is, therefore, not as suitable for programming networks.

An *agent* is like a process except that, while running, an agent can specify and change the processor on which it is executed. Having the identity of processors be under program control is attractive for a variety of technical reasons:

**Communications bandwidth can be conserved.** An agent can move to a processor where data is stored, then filter or otherwise digest that raw data, and finally move on, carrying with it only some relevant subset of what it has read. This means that the computation can be moved to the data rather than paying a cost (in communications bandwidth) for moving the data to the computation.

* Supported in part by ARPA/RADC grant F30602-96-1-0317, NASA/ARPA grant NAG-2-893, and AFOSR grant F49620-94-1-0198. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

**Efficient and flexible service interfaces become practical.** An agent can move to the processor where a server is executing and invoke server operations there using procedure calls. Since the overhead of invoking an operation is now low, the use of agents allows server interfaces having more-primitive operations. Sequences of these operations would be invoked to accomplish a task. In effect, the agent dynamically defines its own high-level server operations—high-level operations that can be both efficient and well suited for the task at hand.

Besides these technical benefits, agents also provide an attractive architecture for upgrading fielded systems with new software. Users of extensible software systems have less incentive to abandon that software, so software manufacturers regard extensibility as critical for preserving their customer bases. Today, it is not unusual for a web browser to download "helper applications" which enable working with new types of data, and much PC software is installed and upgraded by downloading files over the Internet. The logical next step is an architecture where performing an upgrade does not require an overt action by the user. Agents can support such an architecture.

Engineering and marketing justifications aside, agents also raise intriguing research questions. The agent abstraction extends the process abstraction by making processor identities explicit. At first, this might seem like little more than a cosmetic change. But it actually alters the computational model in fundamental and scientifically interesting ways. For example, coordinating replicas of agents in order to implement fault-tolerance requires solving problems that do not arise when it is (only) processes that are being replicated. Another set of new challenges follows from the new forms of sharing and interaction that agents enable, because they are accompanied by new forms of abuse.

These two examples—fault-tolerance and security—are the subject of this paper. Specifically, some initial work on agent-replica coordination [10] is discussed in section 2; see [11] for a somewhat expanded treatment. And, section 3 discusses new security work that was motivated by the agent abstraction.

## 2 Agent Fault-tolerance

An agent, roaming a network, is easily subverted by a processor that is faulty. The most challenging case is when no assumptions can be made about the behavior of faulty processors. Then, executing on a faulty processor might corrupt an agent's program and/or data in a manner that completely alters the agent's subsequent behavior. Moreover, a faulty processor can disrupt an agent's execution, even if that agent is never executed there, by taking actions that the processor attributes to that agent or by generating bogus agents that then interfere with the original agent.

Replication and voting can be used to mask the effects of executing an agent on a faulty processor. However, faulty processors that do not execute agents can confound a replica-management scheme—by spoofing and by casting bogus votes—unless votes are authenticated. This authentication can be implemented by hardware, as done in triple modular redundancy (TMR) [16], where each component that can vote has an independent and separate connection to the voter. But, in a network, there may well be no correspondence between the physical communications lines, the replicas, and the voter. Authentication of votes must be implemented in some other manner.

We describe execution of an agent in terms of a *trajectory*, the sequence of processors on which that agent is executed. Some terminology in connection with trajectories will be convenient: the $i^{\mathrm{th}}$ processor in a trajectory is called the $i^{\mathrm{th}}$ stage, the first processor in a trajectory is called the *source*, and the last processor in a trajectory is called the *destination*. (The destination often is the same as the source.) A trajectory can be depicted graphically, with nodes representing processors and edges representing movement of an agent from one processor to another. For example, in Figure 1, $S$ is the source and $D$ is the destination.

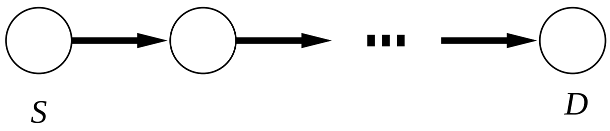


**Fig. 1.** A simple agent computation

The computation depicted in Figure 1 is not fault-tolerant. To tolerate faulty processors (other than the source and destination) in a trajectory, we require that the behavior of the agent at each processor be deterministic[2] and we replicate each stage except the source and destination:

- Each processor in stage $i$ takes as its input the majority of the inputs that it receives from the replicas comprising stage $i-1$.
- Each processor in stage $i$ sends its output to all of the processors that it determines comprise stage $i+1$.

The computation no longer involves a single agent; Figure 2 shows the trajectories of these agents.

Missing from this description is how the voters in stage $i$ processors determine their *electorate*, the processors comprising stage $i-1$. Without this knowledge of electorates, a sufficiently large number of processors could behave as though they are in the penultimate stage and foist a majority of bogus agents on the destination. If agents carry a *privilege*, bogus agents can be detected and ignored

[2] This determinacy assumption can be relaxed somewhat without fundamentally affecting the solution.

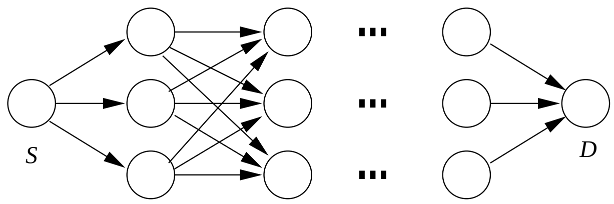


**Fig. 2.** Replicated-agent computation with voting

by the destination. Two protocols for implementing such privileges are sketched in the following.

### Protocols based on Shared Secrets

One implementation of the privilege employs a secret that is initially known only to the source and destination. It is necessary to defend against two attacks:

- A faulty processor that learns the secret might misuse it and direct a bogus agent to the destination.
- A faulty processor could destroy the secret (making it impossible for the computation to terminate).

To simplify the discussion, we start with a scheme that only prevents misuse of the secret by faulty processors. This scheme ensures that, provided a majority of agent replicas in each stage are executed by correct processors, only the source and destination can ever learn the secret.

Clearly, agent replicas cannot simply carry copies of the secret, since the secret could then be stolen by any faulty processor that executes a replica. It is tempting to circumvent this problem by the use of an $(n, k)$ threshold secret sharing scheme [15] to share the secret embodying the privilege.[3] In a system where each stage has $2k-1$ processors, the source would create fragments of the secret using a $(2k-1, k)$ threshold scheme and send a different fragment to each of the processors in the next stage. Each of these processors would then forward its fragment to a different processor in the next stage, and so on.

This protocol is flawed. Minorities in different stages can steal different subsets of the secret fragments. If together these minorities hold a majority of the secret fragments, then the faulty processors can collude to reconstruct the secret. One way to stop collusion between processors separated by a vote is to further divide the secret fragments after each stage. Here is the outline of a protocol based on this insight. For a system with $2k - 1$ processors per stage:

[3] In an $(n, k)$ *threshold scheme*, a secret is divided into $n$ *fragments*, where possession of any $k$ fragments will reveal the secret, but possession of fewer fragments reveals nothing.

- The source divides the secret into $2k-1$ fragments using a $(2k-1, k)$ threshold scheme and sends each fragment along with an agent to one of the processors in the next stage.
- A processor in stage $i$ takes all the agents it receives from stage $i-1$, extracts the secret fragment in each, concatenates those fragments, and divides that into $2k-1$ fragments using a $(2k-1, k)$ threshold scheme. Each fragment is then sent along with the agent to a different processor in stage $i+1$.
- The destination uses the threshold scheme (backwards) to recover the original secret.

This protocol is immune to faulty processors that might misuse a secret, the first of the attacks described above. To address the second attack—the destruction of the secret by a faulty processor—it is necessary to replace the secret sharing scheme with verifiable secret sharing (VSS) [4]. VSS schemes enable correct reconstruction of a secret (or uniform discovery that no secret was distributed) even when processors, including the source, are faulty.

The protocols just described are inefficient, because secret sharing schemes require the aggregate size of the fragments to grow by a constant fraction. However, a protocol is described in [11] that works with constant-size messages; it employs the same basic ideas as outlined above, but with proactive secret sharing [6].

**Protocols based on Authentication Chains**

A second scheme for implementing the privilege is based on pedigree: all processors are assumed to know *a priori* the identity of the source, agents carry (unforgeable) certificates describing their trajectories, and voters use that information in order to reject bogus agents.

Let $[A]_p$ denote a text $A$ that has been cryptographically signed by processor $p$. A digital certificate $[(p:P) \to (q:Q)]_p$, called a *forward*, will accompany the agent that a processor $p$, a member of electorate $P$, sends to processor $q$, a member of electorate $Q$. Notice that if $p$ is non-faulty, then $p$ is the only processor[4] that can construct forward $[(p:P) \to (q:Q)]_p$; any processor can check the validity of such a certificate, though.

To simplify the exposition, assume that processors in distinct stages are disjoint and that the source initiates only this one agent computation. Let $P_i$ be the set of processors comprising stage $i$. As above, we assume that a majority of the processors in each stage are non-faulty. The voter at a processor $q$ in stage $i+1$ can select some agent that was sent by a non-faulty processor $p$ of stage $i$ if two things hold: (i) that voter knows its electorate $P_i$ and (ii) processor $q$ can authenticate the sender of each message that $q$ receives. The selection would then be done by the voter as follows.

[4] Recall that we are assuming that faulty processors can collude. Therefore, a faulty processor can forge a forward on behalf of any other faulty processor.

- The voter uses sender-authentication to reject any agent that is not from a processor in electorate $P_i$—this foils attacks by faulty processors that are not in the previous stage.

- Of those agents that remain, the voter selects any agent for which it has received $\lceil \mid P_i \mid /2 \rceil$ equivalent[5] replicas—this masks the effects of faulty processors from the previous stage.

Sender authentication can be implemented with digital signatures. Thus, all that remains is to describe a means for the voter at $q$ to learn electorate $P_i$.

The processors in stage $i-1$ necessarily know what $P_i$ is, because any agent executing on a stage $i-1$ processor is supposed to be sent to all processors in $P_i$ (i.e. stage $i$). Therefore, $P_i$ becomes available to $p$, to $q$, and to all processors in stage $i$ and beyond, if each agent carries with it the forward $[(r : P_{i-1}) \rightarrow (p : P_i)]_r$ when sent by a processor $r$ at stage $i-1$ to a processor $p$ of stage $i$.

Processor $q$ receives agents carrying these forwards from processors in $P_i$, but might also receive an arbitrary number of agents carrying other forwards due to the activities of faulty processors. If the bogus agents can be detected, then the voter at $q$ can determine electorate $P_i$ by discarding the bogus agents and taking the majority value for $P_i$ in the remaining forwards.

The bogus agents can be detected if agents carry a sufficient number of forwards. Specifically, an agent will carry enough forwards so that a voter can check that the trajectory of the agent started at the source and passed through a sequence of stages, where a majority of the processors in each stage agreed on the electorate of the next stage and agreed on what agent to send to the processors in that stage. And these additional forwards would be carried by agents, provided that each voter augments the set of forwards associated with the agent it selects to be executed. The forwards of all agents in the majority are added by the voter to the set of forwards already being carried by the agent that is selected and executed. Faulty processors cannot produce an agent having such a set of forwards.

The protocol just developed is summarized below; notice that message size now grows linearly with the number of replicas and stages.

- Source processor $p$ sends agents to each of the processors in $P_2$, the processors comprising stage 2. The agent that is sent to a processor $q$ carries the forward $[(p : P_1) \rightarrow (q : P_2)]_p$.

- The voter at each processor $p$ of stage $i$:
  - receives agents from processors comprising stage $i-1$ (and perhaps from faulty processors elsewhere),

  - discards any agents that do not carry a suitable set of forwards. Such a set contains forwards to establish a trajectory that started at the source and

[5] Replicas of an agent are considered to be *equivalent* if they differ only in the sets of forwards they carry.

passed through a sequence of stages, where a majority of the processors in each stage agreed on the next stage and agreed on what agent to send to the processors in that stage.

- determines whether a majority of the remaining agents are equivalent and, if so, augments that agent's set of forwards to include the forwards of all agents in the majority.

– When an agent is ready to move from a processor $p$ to next stage $i + 1$: for each processor $q$ in the next stage, the forward $[(p : P_i) \rightarrow (q : P_{i+1})]_p$ is added to the set of forwards carried by that agent, and the agent is digitally signed and sent by $p$ to $q$.

## 3 Security through Automata

Not only must agents be protected from attack by processors, but agents must be protected from attack by other agents. And, processors must also be protected from attack by malevolent agents. Section 2 concerns protecting against attacks by processors. In this section, we turn to the other piece of the problem—attacks by other agents.

A *security mechanism* prevents executions that, for one reason or another, have been deemed unacceptable. Typically, a security mechanism will work by truncating an execution that is about to violate the *security policy* being enforced. For example, the security policy might restrict what files users can access and a security mechanism for that policy would maintain information for each file and use that information to block file-access operations whose execution would not comply with the security policy.

With agents, new security policies become desirable—policies where the acceptability of an operation depends on past execution. For example, a policy might stipulate

**Policy**: No message is sent after reading any file. (1)

as a way to prevent information from being leaked. Notice, the per-object permission bits that operating systems traditionally maintain would not suffice for enforcing this policy; historical information must be saved.

### 3.1 Characterizing Security Properties

Formally, a *security policy* is a set of executions. Acceptable executions are included in the set; unacceptable executions are not. The manner in which executions are represented[6] is not important, but the ways in which this set can be characterized is. As we shall see, a practical implementation for a security policy $\mathcal{P}$ exists if and only if the set $\mathcal{P}$ can be characterized in certain ways.

[6] Finite and infinite sequences of atomic actions, of program states, and of state/action sequences each can work.

It seems reasonable to postulate that any mechanism for enforcing a security policy must work by analyzing the single execution in progress. This, because a mechanism's use of more than that execution would imply that the mechanism possessed the capability to analyze program text and determine whether some given instruction ever could execute. And such a capability cannot be programmed, given the undecidability of the halting problem [5].

In [1], a set of executions that can be defined by characterizing its elements individually is called a *property*. Using that terminology, we conclude that a security policy must be a property in order for an enforcement mechanism to exist. Discretionary access control [8] and mandatory access control [3] both are examples of security policies that are properties; prohibition of information flow, however, is not a property [9].[7]

Next, observe that a security mechanism, lacking the ability to predict the future, must be conservative about truncating execution. A mechanism for enforcing security policy $\mathcal{P}$ must prevent any partial execution $\sigma$, where $\sigma \notin \mathcal{P}$. This is because were $\sigma$ permitted, the program being executed could terminate before $\sigma$ is extended into an element of $\mathcal{P}$, and the mechanism would then have failed to enforce $\mathcal{P}$.

We can formalize this *conservatism* requirement for enforceability of security policies as follows. For $\sigma$ a finite or infinite execution having $i$ or more steps, and $\tau$ a finite execution, let:

$\sigma[..i]$ denote the prefix of $\sigma$ involving its first $i$ steps

$\sigma\,\tau$ denote execution $\sigma$ followed by execution $\tau$

Then, the conservatism requirement for a security policy is:

$$\sigma[..i] \notin \mathcal{P} \Rightarrow (\forall \tau : \ \sigma[..i]\,\tau \notin \mathcal{P}) \tag{2}$$

Properties $\mathcal{P}$ satisfying (2) are *safety* properties, the class of properties stipulating that no "bad thing" happens during an execution. Formally, a property $\mathcal{S}$ is defined to be a safety property if and only if it satisfies [7]

$$\sigma \notin \mathcal{S} \Rightarrow (\exists i : (\forall \tau : \sigma[..i]\,\tau \notin \mathcal{S}), \tag{3}$$

and this means that $\mathcal{S}$ is a safety property if $\mathcal{S}$ is characterized by a set of finite executions that are the prefix of no execution in $\mathcal{S}$. Clearly, a property $\mathcal{P}$ satisfying (2) has such a set of finite prefixes—the set of prefixes $\sigma[..i] \notin \mathcal{P}$—so such a $\mathcal{P}$ is a safety property.

We have thus established that a security policy must be a safety property in order for an enforcement mechanism to exist. In the abstract, such an enforcement mechanism is a recognizer for the set of executions that is the security policy. The recognizer accepts exactly those executions that are in the policy.

[7] This means that mechanisms purporting to enforce information flow really enforce something different. Typically what they enforce is a property that (only) implies the absence of information flow by also ruling out some executions in which there is no information flow.

Recognizers for safety properties, then, must form the basis for all extant security policy enforcement mechanisms. Moreover, such recognizers must also be the basis for all new mechanisms, including mechanisms powerful enough to enforce security properties, like "no message is sent after a file is read," that previously have not been of concern.

### 3.2 Automata Specifications for Security Policies

A class of recognizers for safety properties was first defined (but not named) in [2]. Here, we shall refer to these recognizers as *security automata*. Security automata are similar to ordinary finite-state automata [5]. Both classes of automata involve a set of states and a transition relation such that reading an input symbol causes the automaton to make a transition from one state to another.[8] However, an ordinary finite-state automaton rejects a sequence of symbols if and only if that automaton does not make a transition into an accepting state upon reading the final symbol of the input. In a security automaton, all states are accepting states; the automaton rejects an input upon reading a symbol for which the automaton's current state has no transition defined. This acceptance criterion means that a security automaton can accept inputs that are infinite sequences as well as those that are finite sequences.

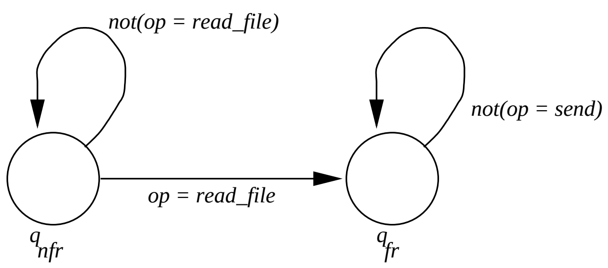


**Fig. 3.** No message sent after reading a file

As an example, Figure 3 depicts a security automaton that recognizes the above policy (1), which prohibits sends after file reads. The automaton's states are represented by the two nodes labeled $q_{nfr}$ (for "no file read" ) and $q_{fr}$ (for "file read"). In the figure, a transition between automaton state $q_i$ and $q_j$ is denoted by an edge linking node $q_i$ to node $q_j$ and labeled by *transition predicate* $p_{ij}$. The security automaton, upon reading an input symbol $s$ when in automaton state $q_i$, changes to the state $q_j$ for which transition predicate $p_{ij}$ is satisfied by $s$.

[8] And, in both classes, non-deterministic automata are defined in terms of transition relations over sets of automaton states.

In Figure 3, the predicate labeling the transition from $q_{nfr}$ to itself is satisfied by system execution steps that are not file read operations; the predicate labeling the transition from $q_{nfr}$ to $q_{fr}$ is satisfied by system execution steps that are file read operations. No transition is defined from $q_{fr}$ for input symbols corresponding to message-send execution steps, so the security automaton rejects inputs in which a send operation follows a file read operation.

As a further demonstration of how security automata describe security policies, consider the two well-known and oft-implemented security policies:

**Discretionary Access Control.** This policy prohibits operations according to a system-maintained *access control matrix*. Specifically, given access control matrix $A$, a subject $S$ is permitted to execute an operation $\Theta(Obj)$ involving object $Obj$ only if $\Theta \in A[S, Obj]$ holds.

**Mandatory Access Control.** This policy prohibits the execution of operations according to a partially ordered set of *security labels* that are associated with the system's objects. The prohibitions prevent information flow from objects assigned high labels to objects assigned lower labels.

For example, a system's objects might be assigned labels from the set

$$\{\text{topsecret}, \ \text{secret}, \ \text{sensitive}, \ \text{unclassified}\}$$

ordered according to:

$$\text{topsecret} \succ \text{secret} \succ \text{sensitive} \succ \text{unclassified}$$

Suppose three types of system operations are supported—read, write, and upgrade. Then, a mandatory access control policy would restrict execution of these operations according to:

(i) a process with label $p$ is permitted to execute a read naming a file with label $F$ provided $p \succ F$ holds.

(ii) a process with label $p$ is permitted to execute a write naming a file with label $F$ provided $F \succ p$ holds.

(iii) an upgrade operation may be executed to change the label associated with an object from $l$ to $l'$ provided $l' \succ l$ holds.

Both discretionary access control and mandatory access control can be described using one-state security automata. The sole state has a transition to itself. For discretionary access control, the transition predicate associated with that transition is:

$$op = \Theta(Obj) \ \wedge \ \Theta \in A[S, Obj]$$

For mandatory access control, it is a predicate concerning security labels that formalizes rules (i)–(iii) above.

It should not be surprising that discretionary access control and mandatory access control can be described as security automata. By construction, security automata characterize security policies that might be enforceable. Security policies, like discretionary access control and mandatory access control, that have actually been implemented certainly are in that class.

### 3.3 Enforcing Policies Defined by Security Automata

How to enforce security policies—and not just how to specify them—is ultimately what is of interest. Therefore, we now turn attention to the enforcement question, describing a general implementation framework and discussing how various mechanisms in the literature can be seen as instances of that framework.

A security automaton can serve as an enforcement mechanism for some *target system* if the automaton can monitor and control the execution of that system. Each symbol (action or new state) that corresponds to the next step that the target system will take is sent to the automaton and serves as the next symbol of the automaton's input. If the automaton cannot make a transition on that input symbol, the target system is terminated; otherwise, the target system is allowed to perform that step and the automaton state is changed according to its transition predicates.

Two mechanisms are involved in this implementation approach:

**Automaton Input Read:** A mechanism to determine that an input symbol is being produced by the target system and then to forward that symbol to the security automaton.

**Automaton Transition:** A mechanism to determine whether the security automaton can make a transition on a given input and then to perform that transition.

The cost of these mechanisms determines the impact of using an enforcement mechanism built from them.

Hardware support often allows Automaton Input Read to be implemented in a way that has minimal impact on performance. For example, when the only inputs to the security automaton are the occurrences of system calls, then we can implement Automaton Input Read by using the hardware-trap mechanism that causes the transfer of control to an operating system routine whenever a system call is made. (The operating system then implements Automaton Transition.) Notice that, with this scheme, no enforcement overhead is incurred for executing other instructions, and since system calls are likely to be infrequent, the overall cost of enforcement is low. Moreover, security policies only involving system calls are rather common. The three security policies discussed in this paper—policy (1), discretionary access control, and mandatory access control—all are instances.

Most operating systems implement some form of memory protection. Memory protection is a security policy. It is a form of discretionary access control, where the operations are `read`, `write`, and `execute`, and the access control matrix tells which processes can access each region of memory. The usual implementation of memory protection employs hardware support: base/bounds registers or virtual memory. This can be seen as a hardware-supported implementation of Automaton Input Read in light of an optimization:

**Automaton Input Read Optimization:** An input symbol is not forwarded to the security automaton if the state of the automaton just after the transition would be the same as it was before the transition.

Legal memory references do not cause the memory protection security automaton to change state, and they are not forwarded to the automaton. Illegal memory references cause traps, and they are forwarded to the automaton.

Another enforcement mechanism that has recently attracted attention is *software fault isolation* (SFI), also known as "sandboxing" [17]. SFI implements memory protection, but without hardware assistance. Instead, a program is edited before it is executed, and only such edited programs are run. The edits insert instructions to test the values of operands, so that illegal memory references are caught before they are attempted.

Cast in our framework, the inserted instructions for SFI simply implement Automaton Input Read by implementing Automaton Transition in-line after each target system instruction that produces an input symbol. The security policy being enforced by SFI is specified by a one-state automaton. However, nothing would prevent a more complicated automaton from being used—say, a multi-state automaton having transition predicates that are richer than the simple bounds checks that comprise SFI. If the security automaton's transition predicates are sensitive to all state changes, in-line code for Automaton Transition might have to be inserted after every target system instruction. A decent optimizer, however, should be able to simplify this added code and eliminate the useless portions.[9]

Finally, there is no need for a run-time enforcement mechanism if the target system can be analyzed and proved not to violate the security policy of interest. This approach is employed for the same kinds of security policies that SFI addresses—policies specified by one-state security automata—with *proof carrying code* (PCC) [13]. In PCC, a proof is supplied along with a program, and this proof comes in a form that can be checked mechanically before running that program. The security policy will not be violated if, before the program is executed, the accompanying proof is checked and found to be correct. The original formulation of PCC required that proofs be constructed by hand. This restriction can be relaxed. For security policies specified by one-state security automata, [12] shows how a compiler can automatically produce PCC from programs written in high-level, type-safe programming languages.

To extend PCC for security policies that are specified by arbitrary security automata, a method is needed to extract proof obligations for establishing that a program satisfies the property specified by such an automaton. Such a method does exist—it was described in [2].

## 4 Concluding Remarks

This paper has touched on two problems that arise in connection with processes that can roam a network. Our purpose was to illustrate that this agent paradigm provides opportunities for rich scientific investigations.

[9] Úlfar Erlingsson of Cornell is currently implementing a system that does exactly this.

Of course, there is more to implementing fault-tolerant agents than a protocol used by voters to authenticate agents comprising an electorate, the problem addressed in section 2. The agent replicas must also somehow find independent processors running the services they require, for example. In addition, the question of ensuring that replicas don't diverge must be addressed. Techniques developed for the state machine approach [14] may not be applicable in this setting, because replicas might execute different sets of requests or execute requests in different orders.

And, there is much work to be done with regard to enforcing security policies in systems of agents. Execution of an agent can span multiple processors. Information about where an agent has executed and what it has done—plausible inputs to a security policy—is not necessarily available to every processor on which the agent executes. The agent itself cannot be trusted to convey such information, nor can the other processors be trusted to furnish it. Perhaps cryptographic techniques will enable some of this information to be conveyed, but it is easy to show that some of an agent's trajectory can be hidden by certain forms of attack.

**Acknowledgments**

I am grateful to my past and current collaborators in the TACOMA (Tromso and Cornell Moving Agents) Project: Dag Johansen, Robbert van Renesse, Greg Morrisett, Úlfar Erlingsson, Yaron Minsky, Scott Stoller, and Lidong Zhou. Robbert's efforts have been instrumental in initiating all of the work described herein.

## References

1. Alpern, B. and F.B. Schneider. Defining liveness. *Information Processing Letters 21*, 4 (Oct. 1985), 181–185.
2. Alpern, B. and F.B. Schneider. Recognizing safety and liveness. *Distributed Computing 2* (1987), 117-126.
3. Bell, D.E. and L.J. La Padula. Secure computer systems: Mathematical foundations. Technical Report ESD-TR-73-278, Hanscom AFB, Bedford, Mass., Nov. 1973.
4. Ben-Or, M., S. Goldwasser, and A. Widgerson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. *ACM Symposium on Theory of Computing*, 1988, 1–10.
5. Hopcroft, J. and J. Ullman. *Formal Languages and Their Relation to Automata.* Addison Wesley Publishing Company, Reading, Mass., 1969.
6. Jarecki, S. *Proactive Secret Sharing and Public Key Cryptosystems.* Master's thesis, MIT, Sept. 1995.
7. Lamport, L. Logical Foundation. In *Distributed Systems-Methods and Tools for Specification*, Lecture Notes in Computer Science, Vol 190. M. Paul and H.J. Siegert, eds. (1985), Springer-Verlag, New York.

8. Lampson, B. Protection. *Proceedings 5th Symposium on Information Sciences and Systems* (Princeton, New Jersey, March 1971), 437–443. Reprinted in *Operating System Review 8*, 1 (Jan. 1974), 18–24.

9. McLean, J. A general theory of composition for trace sets closed under selective interleaving functions. *Proceedings 1994 IEEE Computer Society Symposium on Research in Security and Privacy* (Oakland, Calif., May 1994), IEEE Computer Society, Calif., 79–93.

10. Minsky, Y., R. van Renesse, F.B. Schneider, and S.D. Stoller. Cryptographic support for fault-tolerant distributed computing. *Proc. of the Seventh ACM SIGOPS European Workshop "System Support for Worldwide Applications"* (Connemara, Ireland, Sept. 1996), ACM, New York, 109–114.

11. Minsky, Y. and F.B. Schneider. Agents with Integrity: Tolerating Malicious Hosts. In preparation.

12. Morrisett, G., D. Walker, and K. Crary. From ML to typed assembly language. In preparation.

13. Necula, G. Proof-carrying code. *Proceedings of the 24th Annual Symposium on Principles of Programming Languages* (Paris, France, Jan. 1997), ACM, New York, 106–119.

14. Schneider, F.B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys 22*, 4 (Dec. 1990), 299–319.

15. Shamir, A. How to share a secret. *CACM 22*, 11 (Nov. 1979), 612–613.

16. Siewiorek, D.P. and R.S. Swarz. *The Theory and Practice of Reliable System Design*. Digital Press, Bedford, Mass. 1982.

17. Wahbe, R., S. Lucco, T.E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. *Proceeding of the Fourteenth ACM Symposium on Operating Systems Principles* (Asheville, North Carolina, Dec. 1993), ACM, New York, 202–216.

This article was processed using the LaTeX macro package with LLNCS style