Monotonicity and Partial Results Protection for Mobile Agents*

Bennet Yee†

September 20, 2002

Abstract

Remotely executing mobile code introduces a plethora of security problems. This paper examines the "external agent replay" attack, identifies the notion of one-way program state transitions, describes the use of monotonic variables as a practical method for detecting these attacks, examines the more general problem state modification attacks, and introduces the use of "results verification vectors" to protect agents from attack.

1 Introduction and Motivation

While the remote execution of code eliminates most of the communications latency due to multiple message round-trips needed for repeated, possibly long distance/high latency RPCs, its attraction is greatly diminished by critical security needs: without knowing whether malicious servers have subverted the remote computation or have spied on the agent, how can users trust the obtained results? Though recent theoretical advances [2, 8] show that remote execution can be efficiently verified in principle, the need for reductions via Cook's theorem makes their practical infeasibility obvious.

Mobile agents—a form of mobile code—are vulnerable to attack from the servers upon which they run. A dishonest server can subvert a mobile agent's execution in many ways. Just as a server can lie to client programs in response to an RPC request, agent servers upon which mobile agents run can lie to agents in response to service requests. Agent servers can do much more than simply lie to agents, however, and this makes the mobile agent security problem much more interesting and difficult.[1] An agent server is a powerful adversary: it can examine the internal state of a mobile agent, it can alter that state, it can even subvert the agent's execution on an instruction-by-instruction basis. (Note, of

*This research was funded in part by the National Science Foundation, CAREER Award CCR-9734243; and the Office of Naval Research, Award N00014-01-1-0981.

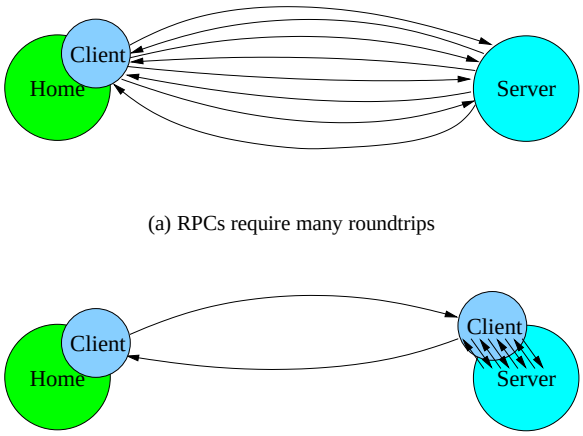
†University of California, San Diego. bsy@cs.ucsd.edu

[1] If the agent merely migrated to an agent server located nearer the RPC service to lower latency costs, the RPC server might be honest but have its messages subverted by the agent server.

course, that the state may be encrypted or otherwise protected, so the server may not "understand" the internal state or know how to modify it profitably.) Indeed, the server can choose to not run the agent at all, but instead make changes to the agent's state in a manner not prescribed by the agent's code, or create a new state out of thin air and pass it to the next server as if it were legitimately computed.

When an agent migrates among the hosts in its itinerary, it may carry data—results derived while executing on earlier servers—into a dishonest server where that data is maliciously modified prior to being carried out to other normal, honest servers. Of course, in general we do not know which of the servers were actually dishonest, and one arm of a multi-pronged mobile agent security defense is to enable the detection of such modifications.

In this paper, we examine the state modification detection problem introduced and addressed in earlier works [12, 32, 17, 33, 24], define two classes of agent replay attacks and describe how such replay attacks can invalidate cryptographic protocols designed for protecting the results of partial computations, examine the general notion of one-way state transitions and focus on how it can be used to protect autonomous multi-hop agents, in particular, against agent replay attacks. (We do not, however, address the question of servers lying to agents.)



(a) RPCs require many roundtrips

(b) Migration has only one roundtrip

Figure 1: RPC versus migration.
When an agent migrates to a server, the RPCs become local requests and do not experience large network latencies.

The next section describes the state modification attack. Next, Section 3 explains the one-way state transition approach for protecting multi-hop agents and analyzes its security properties. We discuss the use of forward secure signatures to protect partial results in Section 4, and then examine an alternative means of protecting partial results—a results verification vector—which provides security even when there are multiple colluding hosts in an agent's itinerary. Section 6 explains the earlier approaches to detecting state modifications, discusses their drawbacks when multiple malicious servers are involved, and examines other security techniques that fit into the one-way state transition framework. Section 7 concludes with a summary and some future directions.

2 Agent Model and State Modificaton Attacks

Mobile agents autonomously migrate to remote servers and perform some computations at each server. In our model, the primary reason for migration is to improve communication efficiency by co-locating the computation with some resource that is available at each of the target servers. Without migration, repeated remote accesses to that resource would have resulted in multiple RPCs and multiple request/response roundtrips—instead of paying for the communications latency many times, agent migration enables efficient, low latency accesses at the cost of a single communications roundtrip delay. Figure 1 shows the long-distance roundtrip reduction of using migration over RPCs.

We assume that distributed computation exhibit remote resource access locality—each of many remote resource is accessed repeatedly and in different phases of the computation. This motivates the use of agents that have a migration itinerary, a list of servers that it visits in turn. Figure 2 illustrates an agent that visits six servers.

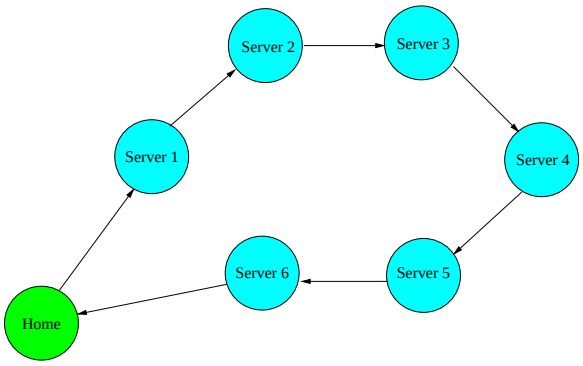


Figure 2: Migrating agents visit several servers in turn.

When an autonomous agent migrates to a server, it carries with it the results of computation performed at earlier servers, and performs queries to some resource controlled by the server while it is there. We model this as follows. Let $y_{i,0}$ be the state that is *input* to server S_i. Based on that state, the agent code generates a query $q_i(y_{i,0})$ to the server resource R_i. The server's response is $x_{i,1} = R_i(q_i(y_{i,0}))$. The agent code computes the next state, $y_{i,1} = f_i(x_{i,1}, y_{i,0})$. If additional resource queries are desired, the agent generates $q_i(y_{i,1})$, obtains response $x_{i,2} = R_i(q_i(y_{i,1}))$, and computes a next state $y_{i,2} = f_i(x_{i,2}, y_{i,1})$, and so on. After k_i queries, the agent code decides that no further queries are needed ($q_i(y_{i,k_i})$ returns a special stop symbol "$\diamond$") and f_i is applied one last time, and the resulting state $y_i = y_{i,k_i+1} = f_i(\diamond, y_{i,k_i})$ is sent to the next server as part of the agent migration.[2] (Note that $k_i = 0$ is possible, in which case the agent is simply using compute cycles on the server.) We denote by $y_0 = y_{1,0}$ the agent's initial state. Figure 3 shows this process in a schematic fashion. We think of y_i as the agent's state after visiting server S_i; it contains the (partial) results derived there which may be used as inputs to later computations done at the next or subsequent servers. If the server is hostile, it may read or write the agent's memory arbitrarily—modify y_i, or the functions q_i, or f_i. Of course, the server may not understand what it reads or writes, e.g., if the memory contains encrypted values, and its modifications might be detected later. This would be the case if the memory values are cryptographically authenticated using digital signa-

[2] We can collapse the multiple rounds of queries to one if the queries can include functional values (or Turing machine encodings) as query parameters; this would simplify the notation by using a form of mobile code (!), but obscures the process.

tures or partial results authentication codes (PRACs) [32, 6, 25] associated with them. In Sections 4 and 5, we will see several proposed cryptographic techniques that aim to protect these partial results.

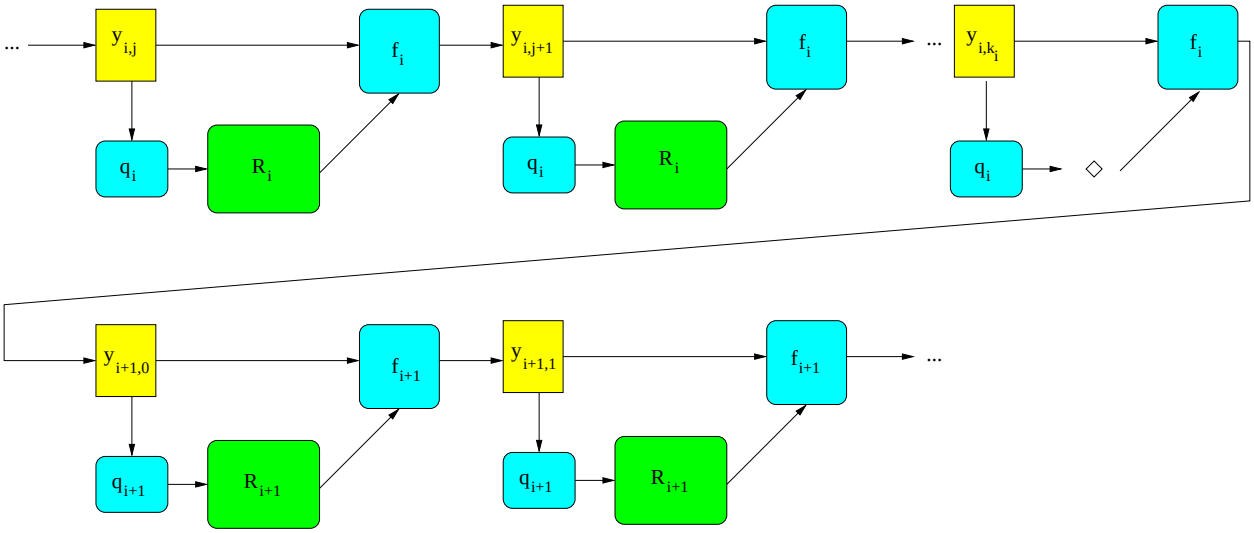


Figure 3: Multi-hop Agent Computation Model
In addition to being multi-hop, the agent makes multiple rounds of queries to the server-side resources on each server. Here each row of computation is performed on a different server.

Note that in this model, an agent server does not have to compromise the resource ($R(\cdot)$) in order to subvert the agent's computation. Indeed, the agent server and the resource may be embodied in machines that are physically located near each other (and so low latency RPCs are possible between them) but in different administrative domains and the RPC service may very well be honest. To obtain the effect of the RPC service lying to the agent, the agent server can subvert the query input, i.e., subverting $q(\cdot)$ so that the obtained answer is to a different query: $x'_{i,j} = R_i(q'_i(y_{i,j-1}))$, or subvert the computation of the next state $y_{i,j} = f'_i(x_{i,j}, y_{i,j-1})$ from the correct resource response $x_{i,j} = R_i(q_i(y_{i,j-1}))$.

Generally we will assume that the implementations of q_i and f_i are immutable code that is cryptographically signed by the agent's author or owner, so while their execution may be subverted on a dishonest server, any code subversion cannot persist to honest servers—but the mutable state y_i is subject to attack. In the Sanctuary security model [14], an agent's code and the initial configuration y_0 are cryptographically signed by the agent's owner, and changes to either would be detected when the agent arrives at an honest server. The signed configuration is typically carried with the agent to later servers, where the agent code will use the configuration to guide—and limit—its actions. We are not concerned with the configuration in this paper, so we will leave it embedded in y_i along with whatever authenticators are used.

2.1 Agent Replay Attacks

2.1.1 Agent Migration Models

Mobile agent systems may implement “strong migration” or “weak migration”. In essence, strong migration systems allow agents to migrate mid-computation—the “place” of the computation (see Section 3.1) is changed as a side-effect of some system request, and the entire agent internal state (possibly excluding external references) is moved to the new machine. In weak migration systems, the agents are restarted from scratch on the new server with different initialization messages; the agent programmers must take care to create the appropriate initialization messages so the next incarnation of the agent will continue from where the previous incarnation left off.

With either form of migration, the agent system implements migration by transmitting the agents’ execution state. In strong migration systems this state is automatically extracted by the system; in weak migration systems the explicit state extraction is the duty of the agent programmer. The extracted state is then sent to the destination host as a parameter of the migration request.

2.1.2 Agent Migration and Replay

Servers receive the state of the migrating agent—abstractly a closure in the strong migration model—perform some computation, and that computation generates the “next state”—another closure—to be passed to the next server in the agent’s itinerary. The message containing the agent state, whether a closure or a hand-crafted representation of the agent’s state, exists in some I/O buffer. Servers are free to copy its value, in essence checkpointing the agent. Because the ability to checkpoint and restore agent state is *inherent* in mobile agent systems, malicious servers have the ability to rerun an agent an arbitrary number of times. While we would like to be able to provide agents with “at most once” execution semantics for executing on servers occuring only once in the agents’ itinerary, the ease with which checkpoints are used would seem to make this impossible.

The rerunning of agents is analogous to message replay attacks that occur in the communications security context. Replay can occur even if the malicious server performs no code analysis, just as an attacker in the network can replay intercepted messages even if the messages are encrypted. In networking, “time to live” fields, sequence numbers, or nonces are used to detect and eliminate duplicate messages. We will see below (Section 3) a secure agent duplication detection mechanism that expands on these concepts.

We divide agent replay attacks into two classes: internal agent replay attacks and external agent replay attacks.

2.1.3 Internal Agent Replay Attack

We define an *internal agent replay attack* to occur when a dishonest server S_i repeatedly runs an agent with different query requests $q_i'(y_{i,j-1}), 0 < j < k_i$ to obtain favorable responses $x_{i,j}' = R_i(q_i'(y_{i,j-1})$ or with modifications to the agent's next-state computation $y_{i,j} = f_i'(x_{i,j}, y_{i,j-1})$ until the output state variable $y_{i,j}, 0 < j \leq k_i$ satisfies some desirability predicate $d(y_{i,j})$, i.e., the server can evolve its attack until a satisfactory output state y_{i,k_i} is obtained. Runs of the agent with unsatisfactory output states are simply discarded. Figure 4 illustrates this attack. Note that only one dishonest server is involved, and there are *no externally observable events* that occur which would make internal replays observable.

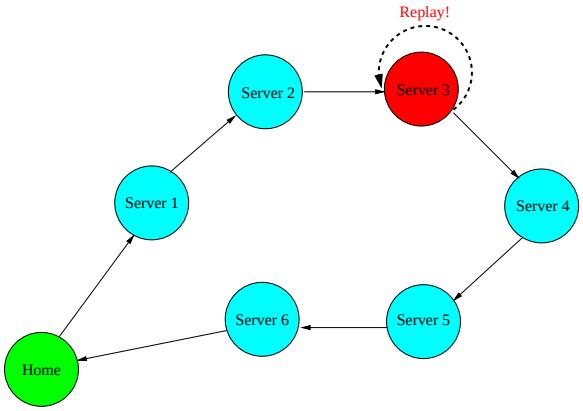


Figure 4: Internal Agent Replay Attack

Agents cannot "remember" having ran before on the dishonest server: in resetting the agent to its earlier arrival state the server also wipes out all memory of previous runs. Whatever memory is used to hold this state information must therefore be external to the agent, and indeed external to the dishonest server. One might imagine that an agent can prevent—or detect—internal agent replay attacks by storing in some secure external memory that execution has commenced and aborting if this is not the first and only time that this has occurred. Without obfuscated code, however, the dishonest server can alter the execution of agent—e.g., by temporarily modifying the agent's code—so that the checks of the external memory values are bypassed.

Note that S_i does not need to "understand" $q_i(\cdot)$ or $f_i(\cdot)$; it merely needs to be able to compute the desirability predicate $d(\cdot)$ to mount this attack. Generally, code obfuscation is impossible [4], though perhaps obfuscating very restricted yet useful classes of functions is still possible. Currently only very limited classes of functions are known to work [27].

Internal agent replay attacks only makes sense for the dishonest server if the server can compute its desirability predicate. This means that if the result of the computation is encrypted somehow (e.g., [8]), servers would be unable to compute $d(\cdot)$ on the agent state to gain (probabilistic) advantage over honestly running the agent. The encrypted function schemes proposed thus far, however, impose heavy performance penalties and are impractical. Practically speaking, then, internal replay attacks seem impossible to prevent or detect.

2.1.4 External Agent Replay Attack

We define an *external agent replay attack* to occur when two or more dishonest servers repeatedly run an agent—through one or more honest servers between them—until the output state variable satisfies some desirability property.

Figure 5 illustrates this process. Note that the case of two or more adjacent colluding dishonest servers—e.g., with no honest servers between them in the agents itinerary—may be analyzed by essentially collapsing the dishonest server nodes into one, i.e., treating the run of dishonest servers as a single dishonest server. This kind of attack is essentially an internal replay attack and is thus impossible to detect.

For external replays that involve one or more intermediate honest server, the situation is not as bleak. We will see in Section 3 solutions where the bracketed honest server helps in detecting external agent replay attacks.

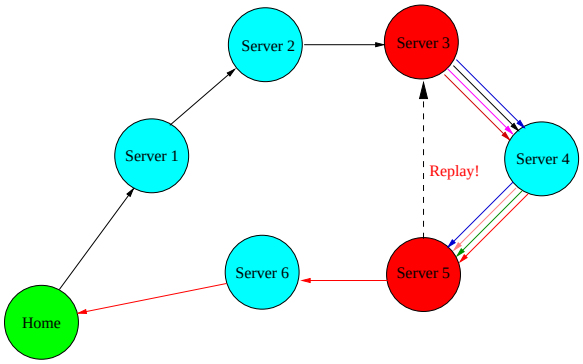


Figure 5: External Agent Replay Attack

2.2 State Modification

Suppose the attacker understands the agent's code or data layout. Such an attacker may, in principle, be merely intercepting and modifying messages in the network, or she may be a malicious server that is in an agent's itinerary. Here, knowing the agent's code or data layout is analogous to knowing partial information about the plaintext that correspond to encrypted messages, and the ability to change that state corresponds to the "malleability" of the encrypted message [11]. Secure mobile agent systems transfer the agent state over encrypted and authenticated channels [26, 32], so an attacker with only network access should be unable to modify a migrating agent's state.

The remaining threat is compromised or dishonest agent servers. Since code obfuscation is, in general, impossible [4] and secure remote computation by reduction to circuits [8] appears to be too expensive, we would like to find alternative practical methods for protecting a multi-hop agent's execution. First, let us describe a simple prototypical attack scenario. Figure 6 sketches the code of an agent which obtains a command while on one server for it to perform one of two actions when at a later server.

While the agent is running on $Server_1$, it obtains additional configuration instructions, perhaps interactively from its user, on what to do later when on $Server_k$. The decision (or the data required to make the decision) is embedded in the state *y1, y2, ...* etc carried to later servers, but to highlight it we separate it out as an extra boolean variable *searchOrDelete*. The threat, of course, is that an intermediate server between $Server_1$ and $Server_k$ is untrustworthy, and can alter the boolean flag *searchOrDelete*. Note that sanity checks on data structures such as state appraisal [12] cannot possibly detect if data values—e.g., the *searchOrDelete* flag—has been changed. Furthermore, those partial results authentication codes which require encrypting the results [25] would make *searchOrDelete* unavailable for use

```
Agent(State y_in)
        migrate(Server_1);
        x = R(query(y_in)); // Server_1 is trusted
        searchOrDelete = predicate(x);
        y_1 = f(x,y_in);
        migrate(Server_2);
        ... /* boolean searchOrDelete modified? */
        migrate(Server_k);
        if (searchOrDelete)
                result = findAllFilesContaining(g(y_in));
        else
                result = deleteAllFilesContaining(g(y_in));
```

Figure 6: Agent flag: Place-Of-Set versus Place-Of-Use vulnerability

Space of possible future execution should be constrained when the flag *searchOrDelete* is set. Since the flag is set at a different server from when the flag is used, intermediate dishonest servers have the opportunity to change its value inappropriately. Declaring variable `final` does not help.

at a later server. We will revisit this scenario when we discuss our proposed solutions.

3 One-Way State Transitions and Monotonicity

One way to look at the agent replay attacks is that we wish to enforce state transition—and time—monotonicity. This is a diametrically opposite goal to that usually encountered in distributed systems where the ability to rollback computation is often desirable [16]. With external agent replay attacks, we detect when a dishonest server rolls back an agent and generate an appropriate error.

The key observation in preventing the replay of mobile agents over their itinerary by malicious servers is that agents cannot depend on state that they carry. Agent state is subject to the control of malicious servers: the replaying malicious server can obviously reset the agent state to an earlier observed one, even if overt state tampering is impossible due to careful sanity checking, state appraisal, or having the agent compute with encrypted values.

If an agent cannot rely on its own record of what has occurred, what can it rely on? Some externally maintained state is needed. We can rely on an outside party accessed via the network (when the agent is running on an honest server, at least; see Section 6.1.2), or we can rely on the honest servers on the replayed itinerary to maintain state. The former case uses a model similar to that in the cooperating agents idea [22, 23], but uses it to solve a slightly different problem: we detect inconsistent agent internal state transitions, rather than just deviations from the agent itinerary or inconsistent final results after computing in disjoint sets of possibly dishonest host. Furthermore, it is immaterial whether we use a trusted server rather than another agent. In a distributed system where high speed

systems make communications cost a dominant factor in the job's time to completion, schemes such as the latter one avoids additional communications steps and thus is more desirable. Before we discuss these two approaches in more detail, we first define what we mean by a one-way state transition.

3.1 Notation

In order to analyze what happens in mobile programs such as that in Figure 6, we must use a slightly more concrete abstraction of an agent program's execution. Instead of consisting of monolithic next state functions $f_i(\cdot,\cdot)$, we must use a model that can incorporate notions of registers, program counters, and memory contents wrapped together in a state variable. While we only informally describe the agent's computation below, how to describe it in formal models such as Turing machines or the RAM model should be clear.

The agent—executing mobile program—consists of an internal state and a *place* or location where it is running. We assume that the migration process merely changes the place; the internal state is preserved. We think of the place as representing properties of where the agent is running, including the namespaces in which external resource names are resolved. The internal state consists of values—memory contents—so all external resources are manipulated by name.

The state transitions that we are concerned with are those that involve changes to the mobile agent's internal state. First, we define the state transition graph.

Definition 1 *A* state transition graph *is a graph* $G = (V,E)$ *where* $V = \{s : s \text{ is a program state}\}$ *and* $(s,t) \in E \subseteq V \times V$*, and there is a transition from state* s *to* t *in some execution of the program.*

A state transition $(s,t) \in E$ occurs whenever there is a state change, e.g., control flow transfer, an ALU operation, or memory modification, that occurs from s to t due to the execution of a single instruction in some execution of the mobile agent's code. At a fine enough granularity, even a `noop` causes a state transition, since the program counter value changes. We are deliberately imprecise about the granularity of the states in this model, since it will be convenient to use this notation at different levels of abstraction of the mobile agent execution.

Definition 2 *A* path *exists from* s *to* t*, denoted* $s \to t$*, when there exists* $j = 0,\ldots,n$ *where* $s_0 = s$*,* $s_n = t$*, and* $\forall i = 0,\ldots,n-1 : (s_i, s_{i+1}) \in E$.

When $s \to t$, we also say that t is *reachable* from s. When no path exists (or $s \not\to t$), we say that t is *unreachable* from s.

Definition 3 *A* one-way state transition *exists from* s *to* t *if* $(s,t) \in E$ *and* $t \not\to s$*. We denote this by* $s \rightsquigarrow t$.

One-way state transitions are interesting to us because their occurence implies an irreversable state change in the (mobile) program—the agent enters a portion of the state space from which it should never escape. In the example of Figure 6, once *searchOrDelete* is set by evaluating predicate(x), its value does not change and what the agent will do on $Server_k$ should be fixed.

Definition 4 *A state x is* inconsistent *with a one-way state transition $s \rightsquigarrow t$ if $t \not\rightarrow x$.*

Definition 5 *A one-way state transition $s' \rightsquigarrow t'$ is* inconsistent *with a second one-way state transition $s \rightsquigarrow t$ if $t \not\rightarrow t'$.*

Definition 6 *Two one-way state transitions $s \rightsquigarrow t$ and $s' \rightsquigarrow t'$ are* mutually inconsistent *if $t \not\rightarrow t'$ and $t' \not\rightarrow t$.*

3.2 External State Transition Verification

By using a program running on an external machine or a secure coprocessor [31] to monitor the execution state transitions, we can detect external agent replay and some state modification attacks. This external program may be a standard service available at many (or all) servers or another (cooperating/monitoring) agent.

The idea here is that the remote service implements a state transition inconsistency detection (STID) algorithm, which records and monitors one-way state transitions that occur during the execution of the agent. As the agent program makes state transitions, messages are sent to the monitor, which then can raise an alarm if any inconsistencies are detected.

For each agent instance (id), the detector maintains a *monotonic* bit $b_{s \rightsquigarrow t} \in \{0,1\}$ associated with every one-way state transition $s \rightsquigarrow t$ in the agent's code:

- When the agent starts, all the bits are created and cleared.
- When the agent reports that $s \rightsquigarrow t$ has occurred, we look for state transition inconsistencies:
  - Examine $b_{s \rightsquigarrow t}$. If it is already set then an agent replay attack has been detected.
  - Examine all bits $b_{s' \rightsquigarrow t'}$ where $s \rightsquigarrow t$ is inconsistent with $s' \rightsquigarrow t'$. If any are set, then a state modification attack has been detected.
- If no attack is detected, set $b_{s \rightsquigarrow t}$.

The salient property is that these one-way state transition monitoring bits are monotonic: once set, they can never be cleared. The monotonicity of their value mirror the one-way nature of the state transition.

In order for STID to correctly detect inconsistencies, the agent must be properly authenticated. Since agents cannot efficiently use cryptographic keys on a server without that server also gaining unfettered access to those keys, note that

a dishonest server can easily cause STID to raise alarms when the agent later executes at an honest server, effectively creating a denial of service "time-bomb." We are not largely concerned with the potential of denial of service attacks by dishonest servers that agents have on their itinerary, since those servers can deny service using simpler means.

In practice, the state space is often too large to be explicitly represented, and a coarser-grained state space representation is used instead, i.e., a state transition may represent the execution of basic blocks of the agent program or even contain entire control-flow structures such as looping constructs. In programs such as Figure 6, we do not care what happens in the agent code between the migrate to $Server_2$ and the arrival to $Server_k$. All that matters is that the state transition the pass of control to the else block:

$$result = \texttt{deleteAllFilesContaining}(g(y_{in}));$$

is inconsistent with the one-way state transition corresponding to setting *searchOrDelete* to zero. Once *searchOrDelete* has been set to zero, that false branch should never be taken. A similar situation exists for the state transitions corresponding to setting *searchOrDelete* to one and the entry to the true branch:

$$result = \texttt{findAllFilesContaining}(g(y_{in}));$$

Similarly, in the case of the external agent replay attack (Figure 5), the one-way state transition of arriving at $Server_4$ must occur only once—trying to set the corresponding bit when it is already set means that arrival to $Server_4$ had occurred once already, and that the current agent is a replayed copy. This check enforces an "at most once" semantics for visiting hosts which occur only once in the agent's itinerary.

While it may seem that a one-way state transition monitor might work for internal agent replay attacks, this is not strictly true. This approach only works if the malicious host cannot "understand" the execution of the agent under its control—otherwise the host can simulate the I/O with the external monitor and fake the transfer of any state transition notifications; such a simulation will not be detected until the host is forced to actually send such a notification, typically prior to letting the agent migrate to another server not under the adversary's control. When there is a run in the itinerary of servers under the adversary's control, none of the state transition notifications need ever be sent to the monitoring agent until the agent leaves the last server in that run.

3.3 Local State Transition Verification

Not all of the checks implemented by STID algorithm have to be performed at a central server. Indeed, the checks may be distributed among the hosts that the agent visits, eliminating extra message roundtrips at the expense of a reduction in coverage. Let us examine this alternative in a little more detail.

Instead of maintaining all the one-way state transition bits $b_{s \rightsquigarrow t}$ at a separate, central server, we would like to maintain some of their values on the agent servers on which the corresponding one-way state transitions occur. This

is not possible for all such bits, but is feasible for many. For example, all the bits corresponding to one-way state transitions of migration arrival may be kept on the agent server, since the arrival event must necessarily occur at the target host of the migrate request. (If the event occurs elsewhere, it is an obvious error or attack.) Similarly, whenever all members of a set of mutually inconsistent one-way state transitions will take place on the same host, the corresponding one-way state transition bits may be maintained on that host. For such a set $S = \{s_i \rightsquigarrow t_i\}$ of mutually inconsistent transitions, we can compress the presentation from a vector of state transition bits to a set-once variable $V_S \in \{\mathsf{undef}, 1, 2, \ldots, |S|\}$, where the value undef corresponds to the all-bits-cleared state and $1 \leq i \leq |S|$ corresponds to when only bit $b_{s_i \rightsquigarrow t_i}$ is set.

Locally maintained one-way state transition monitoring monotonic variables suffices for detecting all external agent replay attacks. Furthermore, it has negligible performance impact—the overhead of migration vastly dominates the cost of looking up, checking, and setting a single bit. Since at least one honest host exists in the replay loop, the honest host(s) will correctly run its portion of the distributed STID algorithm, and the replay will be detected at the first honest server of the replayed portion of the itinerary. For agent server hosts that occur more than once in the agent's itinerary, the agent can maintain a monotonic visit-count variable (perhaps with additional context information) to detect replays. Such a secure visit-count variable serve an analogous role to the time-to-live field in IP packets.

4 Forward Secure Signatures and Monotonicity

Some state transitions such as the setting of final variables [13] are intended to be one-way operations: once a final variable is set, it may never be changed. Contacting a central STID monitor whenever such a variable is set and used would detect inappropriate modifications to its value, but would incur a network traffic (N.B., we need not wait for a response from the STID monitor if we can execute forward optimistically). In this section, we review and analyze the notion of Partial Results Authentication Codes (PRACs) and consider the use of new forward-secure public key signature schemes as an alternative way to secure the values of such variables.

The notion of partial results authentication codes is to have the agent attach an authenticator tag to partial results computed at a server. In our notation, in addition to computing y_i on server $Server_i$, the agent also computes the tag $\tau_i = \mathsf{PRAC}(y_i)$, where PRAC is a possibly stateful partial results authentication algorithm. (In practice only a projection of y_i is used, since not all data encoded in y_i are security relevant.) The tag may be generated using a number of cryptographic schemes, such as forward secure message authentication codes (FS-MACs) [6, 7], or forward secure digital signature (FS-Sig) schemes [3, 5, 1, 20, 15]. The use of FS-MAC schemes for protecting agent computation has been explored elsewhere; here we will concentrate on the use of forward secure signatures.

How forward secure signatures can be used is simple: the generated tag τ_i is simply the signature of y_i using the forward secure signature scheme; when checking the signature, the verifier can determine when the signature was generated, i.e., how many "time periods" had occurred from when the cryptographic keys material were generated. Prior to migrating to the next server, the agent performs the "key update" operation and destroys all copies of the original key. This advances the scheme to the next time period. Thereafter the agent is only able to generate signatures for the next time period—which corresponds to the next agent server—and the agent migrates to the next server with this updated key.

As noted by Hohlfeld et al., [14], there are subtle nuances in the application of forward secure cryptography to mobile agent security. In particular, the protocols needed for transferring keys as part of the agent migration process must provide forward confidentiality, since otherwise long-term keys held by the servers endanger the forward security property of the forward secure cryptographic scheme. This is a concern for FS-Sig schemes as well as forward secure MACs.

Forward secure signature schemes are attractive for generating PRACs. Unlike FS-MACs, the authenticated values may be verified by any later server node in the agent's itinerary. Being public key signatures, the partial results can be verified to be free from malicious state modifications anywhere, without having to migrate "home". For example, this enables mobile agents to perform possibly irreversible actions without having to migrate home for verification, saving extra network roundtrips.

Forward secure cryptography for protecting mobile agents works well when there is only one dishonest host in the agent's itinerary, but provides few guarantees if there are multiple malicious agent servers that could collude to compromise the agent's execution. The important property provided by the "key update" operation of all forward secure cryptographic schemes is that the user *loses the ability to perform certain cryptographic operations,* e.g., to generate tags that would verify as if they were generated during an earlier time period or while running on an earlier agent server. This is a one-way operation—generally the analysis proves that it would be computationally infeasible to recover the value of the keys before the update from the output.

Unfortunately, while one-way operations are involved in the key update, the cryptographic schemes rely on correctly erasing all earlier versions of the keys. A dishonest agent server, of course, does not have to allow an agent to properly run a key update algorithm, and can retain copies of old keys. Since key update algorithms are deterministic, such a malicious server can generate all the keys used in on later servers. When there are more than one dishonest servers on the agent's itinerary that collude, all partial results generated at honest servers situated between the dishonest servers are subject to attack.

5 Protecting Partial Results

The more general problem of protecting the internal state of mobile agents includes protecting all relevant state, not just final variables. Partial results (encoded in y_i) computed at server i, need to be secured in such a way that both the agent's owner and later servers which need to use those partial results.

The inability of forward secure cryptographic schemes to provide strong protection when there are repeated nodes or colluding nodes in the agent itinerary has motivated us to explore alternative partial results authentication schemes that require a slightly different set of assumptions.

The design of some partial results authentication schemes [32] avoided the use of any long-term cryptographic keys held by the agent servers; this removes the incentive for an attacker to try to compromise an honest server after the agent had migrated away from it in order to gain the use of long-term keys that would enable the attacker to subvert the agent currently running on his or her server. This minimizes the motivation time window for mounting attacks. We would like to preserve this property.

Below, we assume that agent servers possess cryptographic keys—they would have to in order to use SSL/TLS to protect the agent migration process in any case—and some of these keys are used by a forward secure encryption scheme [3, 9, 19].

5.1 Notation

In our exposition below, $h(\cdot)$ denotes a cryptographic hash function, $\mathsf{FSE}_k(m)$ denotes the forward-secure encryption of message m using key k, and $\mathsf{FSE}_{k^{-1}}(c)$ denotes the forward-secure decryption of ciphertext c. We are largely unconcerned with key updates, and while the key update period must be globally known, we will omit the key update operations in our description. "Normal" public-key encryption of message m with public key k is denoted $\{m\}_k$, and decryption of cipertext c with the corresonding private key k^{-1} is denoted $\{c\}_{k^{-1}}$. We will abuse notation slightly and denote public key signature—with message recovery—as $\{m\}_{k^{-1}}$—and the verification operation as $\{\sigma\}_k$ which returns either the verified message m where $\sigma = \{m\}_{k^{-1}}$, or $\perp$ if verification fails. (For example, for RSA signatures, the signature $\{m\}_{k^{-1}}$ denotes $(m, h(m)^d \bmod n)$, where h denotes the hash/encode operations, d the secret exponent, and n the modulus.) If a the signature scheme outputs a "bare signature" (from which message recovery is not possible), we use $k^{-1}(m)$ to denote it. (In the example, this would be $h(m)^d \bmod n$.)

We will be somewhat liberal about the security of the functions, i.e., h is collision resistant, FSE is non-malleable, etc.

5.2 Protecting Partial Results

An agent's identity id is a securely compressed version of its *natural name*, $\mathsf{nn} = \{\mathsf{code}, \mathsf{config}\}$, where code is the code and config is the read-only data, including all initial configuration information and a owner-supplied nonce for the agent. The natural name is long and unwieldly, so we typically use $\mathsf{id} = h(\mathsf{nn})$.

5.2.1 Set Up

Agent owner O uses his or her signature key k_O^{-1} to attach to the agent a results verification vector:

$$v = \{\mathsf{id}, k_1, \mathsf{FSE}_{S_1}(\mathsf{id}, k_1^{-1}), k_2, \mathsf{FSE}_{S_2}(\mathsf{id}, k_2^{-1}), \ldots, k_n, \mathsf{FSE}_{S_n}(\mathsf{id}, k_n^{-1})\}_{k_O^{-1}} \quad (1)$$

Note that even though v is read-only, it cannot be part of the agent's read-only configuration config—the value of config is input to the one-way hash function that outputs id, and it should be impossible to solve this recurrence. Because of this, config should contain the owner's identity O or the owner's signature verification key k_O; otherwise it would be easy for an attacker to "take over" a running agent and become its new owner.

5.2.2 Before Leaving a Server

After an agent has completed its partial computation at the i^{th} server in the agent's itinerary, it needs to authenticate the partial results generated at this server. Let y_i denote the partial result.

To perform the authentication, the agent must ask the server to decode v and use its private key S_i^{-1} to decrypt the signature generation key from $c_i = \mathsf{FSE}_{S_i}(\mathsf{id}, k_i^{-1})$. Note that when x_i is decrypted and parsed, the id is checked by the server to prevent a cut-n-paste attack. Because of this, FSE must provide non-malleability—malleability of the ciphertext c_i may enable the adversary to create $c_i' = \mathsf{FSE}_{S_i}(\mathsf{id}', k_i^{-1})$ to enable the decryption key k_i^{-1} to be used with a bogus agent id'.

The corresponding signature verification key, k_i, is not encrypted and is certified by O since v is signed. Note that since this key is used only once, a one-time signature scheme, e.g., based on chains of one-way hash function applications [10], may be used. Next, the agent simply computes $k_i^{-1}(y_i)$ as the partial results signature, and then migrates to the next server.

Agent server hosts destroy its copy of k_i^{-1} when the agent migrates to the next server. However, an attacker that obtains v and breaks in to server i to gain access to S_i can still decrypt k_i^{-1} from v. We require that S_i^{-1} is actually a key used for a forward-secure encryption scheme, and the key update period limits the time period during which a server

compromise is still useful, i.e., $S_i^{-1} = S_i^{-1}(t)$, and after the key is updated it is computationally infeasible to recover the earlier decryption key.

5.2.3 Verifying Earlier Results

Because the signature verification keys are signed by the agent owner O and appear unencrypted in v, verifying partial results generated and signed at an earlier (i^{th}) host in an agent's itinerary is easy: we use v as the last link in a public key certification hierarchy to verify the authenticity of k_i. Given a partial result y_i and its signature σ, we just compute $\{\sigma\}_{k_i}$ after verifying k_i.

5.2.4 Security Analysis

By using FSE, we provide a time bound on the desirability of breaking into a server after an agent has left. While the motivation time window is much larger than that for forward secure cryptographic schemes where key updates occur as the agent leaves, this is a reasonable trade-off for robustness against multiple malicious hosts.

Another drawback for the use of the results verification vector is that in order to properly calculate v, the agent owner must correctly estimate the worse case agent arrival time for each of the servers on the itinerary. The agent owner must balance the risk of late arrival—which makes v useless and therefore would cause the agent to abort—against the risk of having a large time window during which a later malicious server would wish to attack earlier servers in the itinerary.

The construction is relatively straightforward to analyze, and we omit a concrete security style computation of the attacker success probabilities.

The use of STID together with results verification vectors defends against itinerary attacks. We noted earlier [32] that by asking the current server host its identity and the identity of the host from which the agent arrived, agents can ensure that deviations from the itinerary must start and stop at a dishonest host. By checking that the partial results generated at server i are properly signed with k_i^{-1}, we ensure that all hosts in the itinerary are visited at least once. By the use of the STID monitoring bit for the arrival event, we have ensured that each host in the itinerary is visited at most once. Thus, all honest hosts in the itinerary must be visited, though without additional checks the order in which maximal all-honest-hosts segments are visited may be swapped.

6 Related Work

There are three broad areas of related work. First, we describe techniques used to monitor or check agents for signs of tampering. Next, we describe other systems security techniques that bear some similarity to the monotonic variables used in the one-way state transition framework. Finally, we discuss techniques for protecting the results of remote execution.

6.1 Techniques for Monitoring/Checking Agents

6.1.1 State Appraisal

Because a hostile server can freely modify an agent's memory and create inconsistency in the agent's data structures, Farmer et. al. suggested the idea of "state appraisal functions" [12], where upon arrival to a new server, an agent performs a self-check to verify that the appropriate invariants are preserved. While the effect of trying to do such a self-check while on a malicious server can easily be nullified, the key observation is that when the agent (eventually) arrives at an honest server, earlier tampering may be detected.

The kinds of verifications depends on the data structures involved, but generally are inexpensive. In addition to structural invariances, however, there are data value sanity checks that are necessarily application specific. For example, verifying that a singly linked list of unsorted integers has not been modified to contain a cycle (e.g., using the rho method) will not detect if all of the values have been changed.

Generalizing the approach of state appraisal, we can imagine various sorts of sanity checking to be done upon arrival to a new, hopefully honest server.

6.1.2 Cooperating Agents

Roth suggested the use of multiple cooperating agents which can verify each other's execution [22, 23]. A critical assumption is that servers are placed into disjoint sets where set members may collude only with other members of that set. This strategy is effective only if this requirement can be realistically met. In order for accurate partitioning to work, reliable data about the collusion group membership is needed. Such information is likely to be inherently unreliable, and require a trusted outside source of security information.

Earlier Schneider proposed a voting-based scheme [28], where several copies of an agent independently compute and compare their results at intermediate stages. Here the security assumption is that only a fraction of the agent servers may be compromised. This approach eliminates the need for *a priori* knowledge about the trustworthiness

of the servers, but the independence assumption means that it cannot deal with common-mode security failures, e.g., when all servers with a certain type of resource are dishonest.

6.1.3 Privilege Separation

Provos' privilege separation [21] idea gives protection against privilege escalation attacks by separating out code that requires privileges into a separate process. This idea was applied to OpenSSH to defend the `sshd` server process against unforeseen privilege escalation attacks.

The main idea is that the original large privileged process is now separated into two: a privileged process and an unprivileged one. The unprivileged process is still large and potentially subject to attack, and the privileged process monitors the execution of the unprivileged process. The monitor incorporates a hand-crafted finite-state machine (FSM) model of the unprivileged process's execution, and requests from the untrusted program for privileged actions are first checked for consistency with the FSM model. If a request that is inconsistent with the FSM model is detected, the unprivileged process must have been compromised.

In STID, the one-way state transition monitor bits $b_{s \rightsquigarrow t}$ serve to detect inconsistencies in the agent's execution, in much the same way that the FSM is used to detect inconsistencies in privilege separation. Unlike privilege separation, STID is a general service used by mobile agents to perform self examination: agents only need to detect data tampering performed at earlier dishonest servers, since we can safely assume that the agent code and configuration data are unmodified.

6.2 Giving Up Abilities

The following techniques share the common theme of "giving up the ability to perform operations". The monotonic variables used with STID have similar properties: by setting a bit corresponding to a one-way state transition, the agent effectively gives up the ability to make that transition again.

6.2.1 Forward Secure Cryptographic Schemes

All forward secure cryptographic schemes involve some form of "key update" operation, where the ability to decrypt, sign, or MAC data is reduced in some way. Generally, this reduction in ability divides time into distinct periods and protects the user of these schemes, since a later key compromise cannot invalidate operations performed earlier. In the case of FS-Sig, for example, signatures known to have been created prior to the compromise remain valid, since the compromised key cannot be used to forge signatures from earlier time periods.

6.2.2 Physically Protected Memory

In the IBM 4758 secure coprocessor, access to battery-backed secure RAM is carefully controlled [29]. Not only would the 4758 self destruct by erasing all secure RAM contents if any attempt at physical penetration and measurement is detected, the secure RAM is logically protected by carefully controlling access to its contents. Typically the secure RAM contains cryptographic keys. An important goal of the 4758 design is to not only prevent a badly behaving application from divulging secure RAM contents, but to also offer protection against kernel-level compromises from disclosing the secure RAM contents.

The strategy employed in the 4758 design is simple and elegant. A hardware "ratchet" is included within the tamper-detection envelope which controls access to the secure RAM. After a complete reset, all of the secure RAM is accessible. However, as the boot stages progressed, the access ratchet is advanced and access to the more sensitive portions of the secure RAM is withdrawn. The hardware access ratchet is designed to advance monotonically, and the only way its value may be reset is through a power-on reset.

6.3 Protecting Results

6.3.1 Ajanta

Our results verification vector provide a similar service to that of Ajanta's TargetedState mechanism [18], delivering confidential data to be used at a particular server. The Ajanta design leaves the data and its use unspecified, and did not take advantage of this for results verification. Furthermore, the TargetedState encrypted objects are subject to a cut-and-paste attack, where the encrypted objects can be copied into a TargetedState object for a completely different (malicious) agent, the sole purpose of which is to return the decrypted object once the Ajanta server has decrypted it.

In addition to binding the encrypted signature keys to the agent, our results verification vector plays double duty as a link in a certification chain, enabling partial results to be verified at any time.

6.3.2 Proofs of Correctness

Remotely evaluated results can theoretically be verified [2, 8]. While these techniques are efficient in the sense that they involve polynomial time algorithms, they are not efficient in practical terms.

7 Conclusion

The ideas of monitoring one-way state transitions and of results verification vectors improves the chances that attacks by dishonest agent servers will be detected. These techniques show that greater detection sensitivity is possible in practice, without unacceptable performance losses. For external agent replay attacks, a simple distributed variant of the STID algorithm suffices to detect all such replays with negligible overhead. To protect partial results, a combination of using STID and results verification vectors can provide protection at varying levels of granularity: STID permits very fine-grained tampering detection, incurs a great deal of network overhead, but adds little latency if the computation can run forward opportunistically; results verification vectors, on the other hand, provide inexpensive but coarse grained results tampering detection. Neither STID nor verification vectors provide any privacy of remote computation.

To make STID more generally practical, algorithms and tools must be developed to find appropriate one-way state transitions that are security relevant. There has been relatively little work on forward secure public key encryption schemes, and improvements would be desirable.

The performance gap between theoretical results [2, 8] and practical protection schemes is quite large, and additional work to develop improved practical schemes is still needed.

References

[1] Michel Abdalla and Leonid Reyzin. A new forward-secure digital signature scheme. In T. Okamoto, editor, *Advances in Cryptology – ASIACRYPT*, volume 1976 of *Lecture Notes in Computer Science*, pages 116–129. Springer-Verlag, Berlin Germany, December 2000.

[2] William Aiello, Sandeep Bhatt, Rafail Ostrovsky, and S. Raj. Rajagopalan. Fast verification of any remote procedure call: Short witness-indistinguishable one-round proofs for NP. In U. Montanarai et al., editor, *Proceedings of International Colloquium on Automata, Languages and Programming*, pages 463–474, 2000.

[3] Ross Anderson. Two remarks on public key cryptology. Unpublished. Available from `http://www.cl.cam.ac.uk/users/rja14/`.

[4] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *CRYPTO2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2001.

[5] Mihir Bellare and Sara Miner. A forward-secure digital signature scheme. In M Wiener, editor, *CRYPTO99*, volume 1666 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[6] Mihir Bellare and Bennet Yee. Forward integrity for secure audit logs. Technical report, Computer Science and Engineering Department, University of California at San Diego, November 1997.

[7] Mihir Bellare and Bennet Yee. Forward security in private key cryptography. Technical Report 2001/035, Cryptology ePrint Archive, 2001. Available at `http://eprint.iacr.org/2001/035/`.

[8] Christian Cachin, Jan Camenisch, Joe Kilian, and Joy Müller. One-round secure computation and secure autonomous mobile agents. In Ugo Montanari, Jos P. Rolim, and Emo Welzl, editors, *Proc. 27th International Colloquium on Automata, Languages and Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 512–523. Springer-Verlag, 2000.

[9] Ran Canetti and Shai Halevi. Simple forward-secure encryption. Unpublished manuscript available at `http://researchweb.watson.ibm.com/people/s/shaih/pubs/forward.html`, May 2002.

[10] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-26(6):644–654, November 1976.

[11] Danny Dolev, Cynthia Dwork, and Moni Naor. Non-malleable cryptography. In *Proceedings of the 23rd Symposium on Theory of Computing, ACM STOC*, pages 542–552, 1991.

[12] W. M. Farmer, J. D. Guttman, and Vipin Swarup. Security for mobile agents: Authentication and state appraisal. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, pages 118–130, September 1996.

[13] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification, Second Edition*. Addison-Wesley, 2000. Also available at `http://java.sun.com/docs/books/jls/index.html`.

[14] Matthew Hohlfeld, Adijya Ojha, and Bennet Yee. Security in the sanctuary system. Submitted to ICDCS, September 2002.

[15] Gene Itkis and Leonid Reyzin. Forward-secure signatures with optimal signing and verifying. In J. Kilian, editor, *Advances in Cryptology – CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 332–354. Springer-Verlag, Berlin Germany, August 2001.

[16] David Jefferson, Brian Beckman, Fred Wieland, Leo Blum, Mike DiLoreto, Phil Hontalas, Pierre Laroche, Kathy Sturdevant, Jack Tupman, Van Warren, John Wedel, Herb Younger, and Steve Bellenot. Distributed simulation and the time warp operating system. In *Proceedings of the 11th Annual ACM Symposium on Operating System Principles*, volume 21, pages 77–93, 1987.

[17] G. Karjoth, N. Asokan, and C. Gülcü. Protecting the computation result of free-roaming agents. In Kurt Rothermel and Fritz Hohl, editors, *Mobile Agents '98*, volume 1477 of *Lecture Notes in Computer Science*, pages 195–207. Springer-Verlag, September 1998.

[18] Neeran Karnik and Anand Tripathi. Security in the Ajanta Mobile Agent Programming System. *Software Practice and Experience*, pages 301–329, January 2001.

[19] Jonathan Katz. A forward-secure public-key encryption scheme. Technical Report 2002/060, Cryptology ePrint Archive, 2002. Available at `http://eprint.iacr.org/2002/060`.

[20] Hugo Krawczyk. Simple forward-secure signatures from any signature scheme. In ACM, editor, *ACM CCS 2000 Conference on Computer and Communications Security*, pages 108–115, New York, NY, November 1–4 2000. ACM Press.

[21] Niels Provos. Preventing privilege escalation. Technical Report 02-2, Center for Information Technology Integration, University of Michigan, Ann Arbor, MI 48103-4943, August 2002.

[22] Volker Roth. Secure recording of itineraries through cooperating agents. In *Proc. 4th ECOOP Workshop on Mobile Object Systems: Secure Internet Mobile Computations*, pages 147–154, Brussels, Belgium, July 1998. INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex (France), Dépot légal 010598/150.

[23] Volker Roth. Mutual protection of co-operating agents. In *Secure Internet Programming* [30], pages 275–285.

[24] Volker Roth. On the robustness of some cryptographic protocols for mobile agent protection. In *Proc. Mobile Agents 2001*, volume 2240 of *Lecture Notes in Computer Science*. Springer Verlag, December 2001.

[25] Volker Roth. Empowering mobile software agents. Manuscript. `http://www.igd.fhg.de/~vroth/papers/vroth02f_ma.pdf`, 2002.

[26] Volker Roth and Mehrdad Jalali. Concepts and architecture of a security-centric mobile agent server. In *Proc. Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001)*, pages 435–442, Dallas, Texas, U.S.A., March 2001. IEEE Computer Society. ISBN 0-7695-1065-5.

[27] Tomas Sander and Christian F. Tschudin. Protecting mobile agents against malicious hosts. In Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science, Start-of-the-Art Survey*, pages 44–60. Springer-Verlag, 1998.

[28] Fred B. Schneider. Towards fault-tolerant and secure agentry. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, Saarbucken, Germany, September 1997.

[29] Sean W. Smith and Vernon Austel. Trusting trusted hardware: Towards a formal model for programmable secure coprocessors. In *Proceedings of the Third USENIX Workshop on Electronic Commerce*, pages 83–98. USENIX Association, 1998.

[30] Jan Vitek and Christian Jensen. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*. Number 1603 in Lecture Notes in Computer Science. Springer-Verlag Inc., New York, NY, USA, 1999.

[31] Bennet S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.

[32] Bennet S. Yee. A sanctuary for mobile agents. In *Proceedings of the DARPA Workshop on Foundations for Secure Mobile Code*, March 1997.

[33] Bennet S. Yee. A sanctuary for mobile agents. In *Secure Internet Programming* [30], pages 261–274.