A Sanctuary for Mobile Agents

Bennet S. Yee

April 28, 1997

1 Introduction

The Sanctuary project at UCSD is building a secure infrastructure for mobile agents, and examining the fundamental security limits of such an infrastructure.

First, what do we mean by "secure"?

An obvious issue is the privacy of computation. With standard approaches for agent-based systems, a malicious server has access to the complete internal state of an agent: software agents have no hopes of keeping cryptographic keys secret.[1]

The privacy of computation is only one aspect of the security picture: the integrity of computation is perhaps more critical. In agent-based computing, most researchers have been concentrating on one side of the security issue: protecting the server from potentially malicious agents. Related work in downloadable executable content (Java [7], Software Fault Isolation [19], Proof-Carrying Code [16, 17], OS extension mechanisms such as packet filters [13], type safe languages [4, 9], etc) all focus on this problem. The converse side of the agent security problem, however, is largely neglected and needs to be addressed: how do we protect agents from potentially malicious servers? Why should we believe that the result returned by our software agents are actually correct and have not been tampered with?

2 Software Agents and Malicious Servers

In agent-based computing, not only do servers fear that agents bring in viruses or attempt to subvert the server, but the agent's user also needs to be able to trust that the agent was not subverted when visiting a series of servers, some of which may be malicious.

An simple example of how such a subversion might occur will make this problem clearer. Let's look at the standard air-fare agent scenario: I need to travel to Washington D.C. to attend a meeting, so I send a software agent to visit servers at all the airlines to query their databases to determine the least expensive airfare from San Diego to Washington D.C., subject to various trip timing, seat preference, and routing constraints. One of the airlines, Fly-By-Night Airlines, runs a server, `www.flybynight.com`, where my agent's code is automatically recognized and "brain-washed": its memory of what other airlines it has visited and what prices it had seen is modified, so that it ends up recommending a "red-eye" flight by Fly-By-Night Airlines, when a less expensive daytime flight offered by another airline would really have been preferred.

[1]Distributed function evaluation approaches may seem to apply, but that requires an unrealistic fault model and is not likely to be ever practical.

3 Partial Solutions / Preliminary Results

How can software agents be protected from malicious servers? This is a critical security problem to be solved if we are to have faith in agent-based computing. In the following sections, we will examine several approaches and discuss their limitations.

3.1 Legal Protection

One approach to the agent security problem is via legal/contractual means. Operators of the servers where agents run promise, via contractual guarantees, that they will keep their servers secure from external attackers and that they will not violate the privacy or integrity of the software agents' computation. No complex cryptographic protocols are required — there are no run-time overhead at all!

Such an approach, however, is not entirely satisfactory: for it to work the ability to detect breaches of contract is still critical. Furthermore, for that detection to be meaningful, tamper-proof logs must be available to serve as non-repudiable evidence of the breach of contract should lawsuits become necessary.

3.2 No Protection

For certain classes of computations, no protection is necessary, and if we are to carefully examine the cost/benefits of providing protection for software agents, this must be examined. What types of computations require no protection? Suppose the result of the computation is easily verifiable, e.g., the existence of an airfare that is below $200. In this scenario, agents may simply replicate and flood-fill all airline servers to make sure that a copy of the agent has run on each server, and each agent copy can send the corresponding flight information if it finds one that costs less than $200. No agent state needs to be transferred at all.

3.3 Fault Tolerance Approaches

In this section, we discuss fault-tolerance-style approaches to the agent security problem. First, we make some general observations on which aspects of agent state are vulnerable to attack, and which aspects may be systematically verified.

3.3.1 Observations

First, note that uncorrupted servers can determine whether agent code and read-only state have been modified: the originator of the software agent can digitally sign the agent code and all read-only configuration variables before dispatching the agent to agent servers, and the agent servers can verify both the origin and the integrity of these aspects of the agent. (Message authentication codes are inappropriate, since potentially malicious agent servers should not share secrets with the originator of the agent.) Other than cryptographic techniques (if any) needed for the secure communication links, for now we will not require the servers to perform any cryptography.

It may seem that agent code signing could be circumvented by a malicious server, since the malicious server could tamper with the agent and then re-sign it with its own key. This approach, however, is thwarted by the following design: agents are constrained to send its results only to the entity that signed them. Thus, conceptually a server that re-signs an existing agent is simply performing two actions at once: denying service to the true originator of the agent, and sending out its own agent, possibly with initial data stolen from the "murdered" agent.

Next, note that the originator can specify the order in which the software agent will visit the airline servers. Abstractly, this is a circuit of the (complete) graph connecting the airline servers, and the originator may chose this circuit at the time of agent dispatch.

At any honest server, the agent code and its read-only state is checked when the agent arrives, so if the malicious server tries to tamper with the agent code or the read-only state the malicious server can not successfully pass the modified agent to an honest server. (Alternatively, the agent code and the read-only state may be considered to be reloaded from the originator by every server.) Furthermore, we assume that the variable agent state is transmitted among servers using authenticated and encrypted channels, so that only the server that is the intended migration target can receive the agent, as long as the agent is starting from an honest server. Thus, the malicious server can not intercept an agent as it migrates from an honest server to another server.

At any server, an agent may query the server's identity. At first glance, this identity could be authenticated via a public key certification chain, with the root certificate embedded as part of the read-only agent state. Note, however, that the use of cryptographic authentication does not really help a software agent to determine the hosting server's identity: since the server has control over the agent's computation, the malicious server may simply cause the program counter to bypass the cryptography-based identity query and force the program to take the conditional branch(es) which corresponds to the desired (falsified) server identity.

In addition to being able to ask for the identity of the current server, the agent may also ask from which server did the agent migrate. Because we assumed that server-to-server communications use authenticated and encrypted channels, servers will know from which server did an agent arrive. If the agent is running on an honest server, both these answers will be correct and they can be used to verify that the agent had migrated on an edge on the intended migration circuit; if the agent is running on the malicious server, these answers may be incorrect and the agent's state may be modified so that it believes it is running on a different server. In the special case where there is exactly one malicious server, this ruse will be discovered when the agent migrates off of this server to an honest server. If there are two or more malicious servers, the first malicious server encountered by an agent can hand it off to any of the other malicious servers in the route the agent is programmed to take. When the agent is passed on to the next (honest) server, the agent is brainwashed to believe it had visited all the servers in the original path between the two malicious servers, thus avoiding discovery. *If software agents are to depart from the route determined at agent dispatch time, such departures must start and end at a malicious server.*

Now, consider what visiting a malicious server can do to a software agent's memory. The read-write state variables of an agent may be completely altered by the malicious server; thus, an agent that has just left the malicious server can not trust any of its memory: All information collected prior to this point it time — including data from servers visited prior to visiting the malicious server — are suspect. Thus, only the results of computation done by those servers from the (maximal) *honest suffix* of the agent's route, assuming that the computation is independent of any input from previous servers, should be trusted.

3.3.2 Server Replication

In [15], Minsky et. al. developed a general method for mobile agent computation security, marrying some ideas from the fields of fault tolerance and cryptography. They propose that servers should be replicated, and that replicated agents on these servers can use voting and secret sharing/resplitting to move from one phase of the computation to the next.

Unfortunately, the fault model assumed in the paper is completely unrealistic: it assumes that replicated servers fail independently. In our Fly-By-Night Airlines example, all replicated `www.flybynight.com` servers are under the administrative control of `flybynight.com`, and malicious attempts to brainwash software agents would occur on all of these servers. And while bribery of individual administrators of replicated servers by an outside adversary might be independent events, bribery of the software engineers responsible for the `www.flybynight.com` Web site is a much more likely scenario. Even if we assume that Fly-By-Night Airlines is trustworthy, replicated servers in the real world are likely to consist of identical hardware running copies of the same software: any security holes found by an external attacker that allows him/her to compromise one of the replicated servers is very likely to permit him/her to compromise all the servers.

3.3.3 Agent Replication

While the general approach proposed by Minsky et. al. fails to be convincing, in certain special cases the fault tolerance style of approach can solve or at least ameliorate the mobile agent security problem. Because server replication does not help to reduce the risk of agent brainwashing, in the following we will assume that there is only one server per administrative/security domain, or when there are multiple servers in a domain, they are indistinguishable.

Consider the case where there is at most one malicious server in our airfare minimization example. Assume that secure communication links exists between the servers, and that the users possess individual certified public keys; servers may use these keys to verify the origin of the software agents. (Secure communication channels may be constructed cryptographically if servers also possess cryptographic keys to authenticated and encrypted data among the parties as needed.) Because we are assuming that there is only one dishonest server, we know that the agent must stay on the circuit prescribed during agent configuration.

Suppose we chose some sequence of servers $S = s_1, s_2, \ldots, s_n$. We configure two software agents A_1 and A_2, where A_1 will travel along S, and A_2 will travel over $S^{-1} = s_n, s_{n-1}, \ldots, s_1$.

Recall that we are assuming at most one malicious server. The all-honest servers case is trivial, so we can ignore that case; henceforth we will assume that there is exactly one bad server. Without loss of generality, assume that server s_i is malicious, and that s_j is run by the airline with the lowest fare ($j \leq i$). Furthermore, we assume that the malicious server will not attempt denial-of-service attacks — it may do so by killing the software agent or by implanting the belief that the lowest fare is offered by some third server which will later repudiate this idea.

First, consider the $j < i$ case. A_1 will encounter the lowest-fare server (s_j) first, and when it arrives at s_i, its memory of the lowest-fare seen-so-far may be altered. When A_1 returns with its result, it will report either s_i as the server with the lowest fare, or some s_k where $k > i$ if the malicious server did not declare a fare lower than one that the agent will see later in its travels.

A_2, on the other hand, will encounter the lowest-fare server after visiting the malicious server. It will report the correct minimum price — since we assume no denial-of-service attacks, the corrupt server will not have made this agent believe that a (false) lower price exists elsewhere — and when A_2 returns to the user, the user will be able to determine the true minimum airfare.

Next, consider the $j = i$ case. When this occurs, the malicious server can alter its price to be just below that of the second lowest price offered and still get business. This corresponds to a Vickery auction or second-price auction,[2] except the situation is upside-down: instead of the highest bidder paying the second highest bid price to obtain the goods being auctioned off, we have the lowest airfare offer selling tickets at the second lowest quoted price. Note that Vickery-style

[2]We do not have sealed bids here since the minimization is done by the agent; an alternative design would be to gather bids encrypted using the public key of the agent originator, preventing servers from knowing each other's prices directly. Of course, servers could send out their own agents to discover such "commodity" prices; this may have to be done through anonymizing proxies if the pricing could depend on the consumer's identity.

price determinations may be a desirable economic design choice anyway, since Vickery auctions are designed to maximize the flow of pricing information so bidders have no economic interest to hedge and not bid (and reveal) the true prices that they are willing to pay.

The above agent-replication approach provided a partial solution for a special case — at most one malicious server — the solution did not quite work "properly" to compute the true minimum airfare: when $j = i$, we could only achieve second-best pricing, where what we obtain is the second-best airfare minus ϵ. Arguably, since airline servers may also send out agents to determine pricing at other airlines — assuming price information can be obtained anonymously or in such a way that we are assured that it is independent of consumer identity (or race or age or ...) – Vickery pricing may be the end effect whenever there is great consumer price sensitivity in any case. Applying some basic cryptographic techniques, however, we can do a little better.

3.4 Cryptographic Approaches

There are three cryptographic techniques that apply or may apply. The first uses per-server digital signatures to vouchsafe partial results; the second is the use of state authentication codes to improve on the fault-tolerance solution above; and the third is the use of probabilistically checkable proofs (PCP) (a.k.a. "holographic proofs") or computationally sound proofs (CS proofs) to show that the computation at the servers ran correctly. Note that these techniques deal with the integrity of the result of computations at various servers — if the privacy of the result is needed, the result can simply be encrypted using the originator's public key.

3.4.1 Digital Signatures

The application of digital signatures to mobile agents is to use the signatures to vouchsafe partial results from computation done while executing in a server. Going back to the airfare minimization example, what we will do is to have each airline server sign a message of the form "this is the best airfare found by your software agent at this server at this time".[3] The message signature is done using the airline's key, so it is non-repudiable and unforgeable. The message may optionally be encrypted by the originator's public key to ensure privacy.

The key observation is this: due to the unforgeability property, a malicious server can not completely brainwash the agent — at worse, it can make the agent forget the lowest airfare, which will be detected when an enumeration of the signatures show that one airline's quote is missing.

By using digital signatures, we will either obtain the true minimum airfare or be able to detect any tampering with the agent. The cost is that the agent state grows linearly with the number of servers visited, where the fault-tolerance approach required constant space (though the result was Vickery pricing rather than true minimum). This is an acceptable overhead for most applications. Like the fault-tolerance approach, however, the digital signature approach is not fully general: it only applies only certain classes of functions where there are intermediate results that can be "compressed" (e.g., in this case, the intermediate result is the best price found on the server — we don't care how much work was done in querying the airline's databases prior to finding this result, and we don't need to prove that it took place and that it ran correctly).

[3]It is important to note that the query as well as the answer is signed and timestamped. Otherwise signed answers to the wrong question, asked by a different agent dispatched by the adversary, could be substituted in place of the expected answer.

3.4.2 Partial Result Authentication Codes

The idea of a Partial Result Authentication Code (PRAC) is very similar to that of a message authentication code (MAC)[2, 10, 11]. Instead of authenticating the origins of a message, we are demonstrating the authenticity of an intermediate agent state or partial result that resulted from running on a server. Similar to MACs, PRACs are cheaper than digital signatures to compute, and have slightly different security properties.

The property that PRACs ensure is *perfect forward integrity*: if a mobile agent visits a sequence of servers $S = s_1, s_2, \ldots, s_n$, and the the first malicious server is s_c, then none of partial results generated at servers s_i, where $i < c$, can be forged. This contrasts with a simple digital signature, where if an attacker compromises the generating host where the signature key is stored, the authenticity of all messages signed with that key becomes questionable. The use of a digital timestamping service [8] can have similar properties, except that in that case a trusted third party (the timestamping service) is required and the granularity of the timestamps limits the maximum rate of travel for the agents — the agent must stay on a server until the next timestamping epoch before migrating to the next server.

Simple MAC-based PRACs To have an agent use simple PRACs, we provide the agent with a list of secret PRAC keys at agent dispatch, with a key per server visited. Before leaving a server for the next, the agent summarizes its partial results from its stay at this server in a "message" back to the agent dispatcher. This message need not be sent back to the dispatcher immediately; instead, it may be carried with the agent as part of its migrating state for later transmission. This may be delayed so that it is "sent" to the dispatcher when the agent returns. Alternatively, these messages may be "batched" and sent when the networking bandwidth is cheap or available, e.g., when the dispatching mobile host has reconnected to the network. To provide integrity, a MAC is computed on this message, using the key associated with the current server; the message, along with its MAC, comprises the PRAC. The critical difference between MACs and PRACs is that after a PRAC is computed, the agent takes care to erase the PRAC key associated with the current server prior to migrating to the next server.

The erasure step provides a very important security property: the partial results from the *honest prefix* servers can not be modified. (This contrasts with the *honest suffix* property from section 3.3.1, where the partial results from honest servers visited after visiting the last malicious server are known to be unmodified.) Suppose an agent A traverses a sequence of servers $S = s_1, s_2, \ldots, s_n$, where at each server s_i a partial result PR_i is computed using key k_i, and servers $s_1, \ldots s_j$ are honest and do not expose the internal state of the agent, then $\forall i, k : i \leq j < k, s_k$ can not forge PR_i, since s_k must know k_i to change PR_i.

MAC-based PRACs with One-Way Functions An obvious enhancement to simple PRACs is to use a single key instead of n PRAC keys and use a m-bit to m-bit one-way function to generate the list of PRAC keys. When the agent is initially sent to server s_1, it contains key k_1. When the agent prepares to go from server s_i to server s_{i+1}, it computes $k_{i+1} = f(k_i)$, where f is a one-way function, and erases all knowledge of k_i. As before, a server s_k can not forge PRACs from previous servers, since it would have to break the one-way assumption to determine the previous PRAC keys or break the MAC function.

More generally, instead of a m-to-m-bit function, an m to r bit one-way function may be used. (Typically $r < m$.) In this case, to obtain k_{i+1} from k_i, we simply use some (perhaps pseudo-random) known $(m - r)$-bit string t and set $k_{i+1} = f(k_i|t)$, where "|" denotes concatenation. If the probability that any algorithm, given y, will find a pre-image $x : f(x) = y$ is at most $2^{-m} + \epsilon$, knowing the last $(m - r)$ bits of the pre-image $x = k_i|t$ should not help: if knowing t gives an algorithm a probability of finding a pre-image of $p > 2^{-r} + 2^{m-r}\epsilon$, then by guessing the value of t values (probability 2^{r-m}), we obtain an algorithm which will find a pre-image with probability $2^{r-m}p > 2^{-m} + \epsilon$, contradicting our one-way assumption.

Publicly Verifiable PRACs The MAC-based PRACs above required that the agent originator maintained a secret key or keys in order to detect tampering with the partial results. An obvious question is whether perfect forward integrity can be provided such that the integrity verification may be public — so that an untrusted intermediate server not sharing the secret key with the originator may nonetheless help detect tampering.

Like MAC-based PRACs, publicly verifiable PRACs are implemented by relying on the destruction of information when agents migrate. Here, we use a digital signature system: when the agent is dispatched, it is given a list of secret signature functions $\mathrm{sig}_1(m), \ldots, \mathrm{sig}_n(m)$, along with usre-generated certificates for their corresponding verification functions $\mathrm{verif}_1(m, s), \ldots, \mathrm{verif}_n(m, s)$. The verification functions would be signed by the user's signature function $\mathrm{sig}_{\mathrm{user}}(m)$.[4] Like simple MAC-based PRACs, we use $\mathrm{sig}_i(m)$ to sign the partial result computed on server s_i, and erase $\mathrm{sig}_i(m)$ prior to migrating to server s_{i+1}.

Similar to one-way function MAC-based PRACs, we can also defer key generation, so that most of it is done on the servers, which presumably have greater resources. Here, the agent is given an initial secret signature function $\mathrm{sig}_1(m)$ and a certified verification predicate $\mathrm{verif}_1(m, s)$; the signature function is used both to sign partial results and to certify new verification functions.

The verification predicate $\mathrm{verif}_1(m, s)$ (and its certificate) is public, and the signature function $\mathrm{sig}_1(m)$ is secret. When the agent is ready to leave server s_1, it signs the partial result r_1 by computing $\mathrm{sig}_1(r_1)$. Next, it chooses (randomly) a new signature / verification function pair from the signature system, $\mathrm{sig}_2(m)$ and $\mathrm{verif}_2(m, s)$, and computes $\mathrm{sig}_1(\mathrm{verif}_2)$ to certify the new signature functions. Lastly, before the agent migrates to server s_2, sig_1 is destroyed.

To use publicly verifiable PRACs, the list of certified verification predicates must be either published and/or carried with the agent. When these predicates are available with the agent, publicly verifiable PRACs enjoy an important property not available with MAC-based PRACs: while at server s_j, the agent can itself verify the partial results obtained while at servers s_i, where $i < j$. In particular, this means that computations that depend on previous partial results can detect any integrity violation of those results — the agent's computation can abort early, instead of having to finish the computation and detecting integrity violation only when the agent results return to the agent originator.

3.4.3 Proof Verification

We would like to get a guarantee that the agent's computation was done according to program specified in the agent. One possibility is to forward the entire execution trace to the originator, who checks it. This however is too costly. We would like to explore the use of holographic proof checking techniques [1].

This is quite a speculative idea. The current approaches are very theoretical. In principle they do help, but the cost in practice of existing solutions is prohibitive. We are considering investigating ways to use the ideas in a more practical way. Let us describe the ideas and issues to see what it is about.

[4]Signing a function $\mathrm{verif}_i(.)$ simply consists of signing the parameters that specify the function; in RSA, it would be the two values e_i, n_i.

Call the program x. Let y denote an execution trace. Define the predicate $\rho(x, y)$ to be 1 if this trace is correct (corresponding to running x) and 0 otherwise. The server does not want to send y. But it can encode y as a holographic proof y'. This has the property that one needs to look at only a few bits of y' to check that $\rho(x, y) = 1$. It is tempting from this to think that the server can just transmit a few bits. But this does not work. The model necessary for holographic proofs is that the verifier have available a fixed, "committed" proof string y' that he can access at will. He will pick a few random positions here and check something. So there is no choice but to transmit y' in entirety. We will not save bandwidth. We will gain something: the verification process is faster. (The verifier receiving y' will perform some quick spot-checks).

A better approach is to use computationally sound (CS) proofs as in [12, 14]. Having constructed the holographic proof y' as above, the server hashes it down via a tree hashing scheme using a collision-resistant hash function h. Only the root of the tree is sent to the originator. This is relatively short, so bandwidth is saved. In addition, certain challenges are implicitly specified by applying an ideal hash function to this root, and the server also provides answers to them. The total communication from server to originator is still small compared to the length of the original execution trace y, yet some confidence in the correctness of y is transmitted!

The tree hashing is actually not impractical. What is prohibitive is constructing the holographic proof y' to which it is applied. This currently calls for application of NP-completeness techniques, including the use of the construction underlying Cook's theorem. What we might hope instead is to find a direct holographic proof for the functions of interest, and then apply tree-hashing.

3.5 Trusted Environments (Secure Coprocessors)

An engineering solution to providing security for software agents is to build a trusted / trustworthy execution environment for the agents. The Sanctuary project will build such an environment to run within a secure coprocessor [20], allowing Java-based agents to run securely; design and implement the agent APIs needed to support mobile Java agents; and develop the technology by which Java-based agents can migrate among unmodified Java interpreters running in an secure-coprocessor environment.

In addition to the basic support for agent execution, the Sanctuary project will develop the trust framework needed for inter-server communications. This necessarily implies having some basic public key infrastructure — we should be able to leverage off of the existing work being done to support SSL [6], PCT [3], and TLS. [5]

4 Trust Models

The issue of trust models is very important to agent-based computing. Agents do not just need a trusted computing base (TCB) — trust may not be so binary in nature. Instead, agents (or their deployers) may decide that it is okay to run in a software-only environment if such an environment is hosted by a well-known and trusted entity, but the use of physical protection to maintain the trustworthiness of a trusted third-party provided execution environment is needed when the environment is hosted by an entity with no reputation to protect and/or where no legal remedies may be obtained.

In Sanctuary, we envision that the trust decision will be made by the agent's software itself. Thus, trust specification is simply an object in Java, and any effectively computable function may

[5] The Internet Engineering Task Force's Transport Level Security group is developing a merged protocol based on SSL version 3 and features from PCT. Such a protocol requires a merchant-side public key infrastructure.

be used. This is similar in spirit with the work of Blaze and Feigenbaum [5], except that by unifying the agent language and the trust specification language, the programmer's work is simplified.

5 Mobile Java

Other approaches to providing mobility to Java programs [18] requires modifying the interpreter. In Sanctuary, we intend to provide a mechanism to migrate Java-based agents that can run on unmodified interpreters. This strategy will enable wider acceptance of mobile agents, leveraging off of the work done by Sun/Javasoft.

6 Proposed Work / Future Work

The Sanctuary project group will examine the important security issues in mobile agent computing. This paper has discussed some preliminary results and directions.

The primary goal will be to build a secure agent environment insofar as it is theoretically feasible. First, we will build a trusted Java agent environment to run within a secure coprocessor and design APIs that permit agents to exist both in a hardware-based secure environment and in a software-only environment unchanged (but permitting security property queries). Next, we will build the necessary software tools to permit Java-based agents to be mobile. Our techniques will enable these agents to run on unmodified Java interpreters; this design approach permits greater acceptance of our work, since no complex installation process will be required, and it will allow our system to track new Java releases more easily. Additionally, we will examine alternative methods for providing security for software agents through fault tolerance and cryptographic approaches (e.g., distributed function evaluation, additional uses of digital signature techniques, etc).

Acknowledgements

The author wishes to thank Mihir Bellare for his invaluable help in preparing this paper.

References

[1] Laszlo Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing*, pages 21–31, New Orleans, Louisiana, May 1991.

[2] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neil Koblitz, editor, *Advances in Cryptology: Crypto '96 Proceedings*, volume 1109 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[3] Josh Benaloh, Butler Lampson, Terence Spies, Dan Simon, and Bennet Yee. The PCT protocol, October 1995.

[4] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the spin operating system. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995.

[5] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, page (to appear), May 1996.

[6] Alan Freier, Philip Karlton, and Paul Kocher. The SSL protocol version 3, December 1995.

[7] J. Steven Fritzinger and Marianne Mueller. Java security, 1996. Published as `http://www.-javasoft.com/security/whitepaper.ps`.

[8] Haber and Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2), 1991.

[9] Wilson C. Hsieh, Marc E. Fiuczynski, Charles Garrett, Stefan Savage, David Becker, and Brian N. Bershad. Language support for extensible operating systems. In *Proceedings of the Workshop on Compiler Support for System Software*, February 1996.

[10] IBM Corporation. *Common Cryptographic Architecture: Cryptographic Application Programming Interface Reference*, SC40-1675-1 edition.

[11] R. R. Jueneman, S. M. Matyas, and C. H. Meyer. Message authentication codes. *IEEE Communications Magazine*, 23(9):29–40, September 1985.

[12] Joe Kilian. A note on efficient zero-knowledge proofs and arguments. In *Proceedings of the Twenty Fourth Annual ACM Symposium on Theory of Computing*, Victoria, British Columbia, Canada, May 1992.

[13] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX Technical Conference Proceedings*, pages 259–269, San Diego, CA, 1993. USENIX.

[14] Silvio Micali. Cs proofs. In *Proceedings of the 35th IEEE Symposium on Foundations of Computer Science*, pages 436–453, Santa Fe, New Mexico, November 1994.

[15] Yaron Minsky, Robbert van Renesse, Fred B. Schneider, and Scott D. Stoller. Cryptographic support for fault-tolerant distributed computing. Technical Report TR96-1600, Department of Computer Science, Cornell University, July 1996.

[16] George Necula. Proof carrying code. In *Proceedings of the Twenty Fourth Annual Symposium on Principles of Programming Languages*, 1997. To Appear.

[17] George Necula and Peter Lee. Safe kernel extensions without run-time checks. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, October 1996.

[18] Mudumbai Ranganathan, Anurag Acharya, Shamik D. Sharma, and Joel Saltz. Network-aware mobile programs. In *Proceedings of the Usenix 1997 Annual Technical Conference*. Usenix, 1997.

[19] Robert Wahbe, Steve Lucco, T. E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the ACM SIGCOMM 96 Symposium*. ACM, 1996.

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