A Security Architecture for Multi-Agent Matchmaking

Leonard N. Foner

MIT Media Lab
20 Ames St, E15-305
Cambridge, MA 02139
foner@media.mit.edu
617/253-9601

Abstract

Using a multi-agent system to match and introduce users who share interests has important advantages, but handling sensitive data involves a number of design challenges in ensuring user privacy. This paper describes many of them, briefly summarizes some relevant cryptographic technology, and uses this technology to demonstrate how to avoid most of the potential privacy problems without unacceptable performance penalties.

Introduction

In general, agents are useful because they understand some aspects of their users' goals and can carry out actions autonomously to fulfill those goals. This may require that any given agent know personal or sensitive information about its user, and that it must be robust against revealing this information to third parties or allowing its actions to be subverted by a malicious interloper into carrying out an undesired action.

As the value of the information or the motivation to inflict damage increases, the possibility of inadvertent information disclosure or subversion of an agent's goals by an attacker are very real. This is particularly important in applications which handle highly personal information, or which handle real money. In the former, not only may people's personal privacy be violated, but careless handling of this data may be tantamount to violating the laws in some countries that mandate the safe handling of personal data. In the latter, there is the potential for significant financial loss, or—in the case of a publicly-traded company, for example—for violations of securities laws related to confidential information.

Multi-agent systems require that the multiple agents collaborate to accomplish their users' collective goals. Such systems must often depend, to a greater or lesser extent, on sources of information obtained from others, and must often "leak" information about their own internal state or the goals of their users in order to interoperate with their peers; this makes security and privacy harder to achieve. Further, a trusted intermediary is often an impractical or unavailable solution, which complicates the problem.

This paper describes a multi-pronged strategy for improving the security of a multi-agent matchmaker system named *Yenta* [6][7]; the system itself and its concomitant security architecture are under active development. The following sections describe:

- the nature of the problem, including an introduction to Yenta's basic mode of operation, the types of attacks expected, design issues in security systems in general, and what problems we are *not* attempting to solve;
- general techniques, mostly cryptographic, for assuring confidentiality and authenticity of data; and
- the nature of the solution, which uses these general techniques to solve some of the problems in making a multi-agent matchmaker.

Readers of this paper are *not* expected to be well-versed in cryptography or to have extensive prior experience in computer security. The cryptographic techniques mentioned herein are used as "black boxes," without proof that they properly implement the functionality described for the "box" and without the mathematical background which underlies them; those who wish to check these assertions may examine the citations where appropriate.

The Nature of the Problem

This section describes, very briefly, Yenta's underlying architecture, then discusses the types of attacks it is likely to see, as well as the problems we are *not* trying to solve.

Yenta

The fundamental application being supported is that of a completely decentralized, peer-to-peer matchmaking agent named Yenta. This section provides only a brief summary of the underlying structure of Yenta; [7] provides much more complete information on the functioning of the matchmaking parts of Yenta.

Yenta is designed to find people with similar interests on the Internet and introduce them to each other. Such introductions can serve as the basis for instant coalitions or discussion groups, and can help mitigate problems such as two individuals just doors away from each other who are working on similar problems—but never knew it because they didn't happen to mention it to each other. Introductions can take forms such as opening an encrypted two-way channel between two pseudonymous Yentas (whereupon their users can simply type at each other), or more complicated, "flirtation-like" negotiations.

Each user runs his or her own copy of Yenta. Each agent determines the user's interests by scanning all the mail and files owned by the user. Each individual message or file becomes part of a set of larger structures called *granules*; each granule represents some discrete interest. Since Yenta works with ubiquitous data, users need not be able to explicitly articulate their own interests (nor need they spend the time to do so). However, the sensitive nature of the data, combined with the resulting introductions performed, motivate most of the security considerations discussed in this paper.

A user who had, say, 30 disparate interests would thus in the best case have about 30 different granules constructed from his or her files. The user may omit certain granules from consideration: for example, many messages about scheduling meetings look alike to Yenta, and most users would rather *not* have an introduction arranged with someone else just because they both happen to have meetings!

For those granules not omitted, any given Yenta follows a *referral algorithm* that finds other Yentas that share interests, as follows. Each Yenta maintains a *cluster cache* and a *rumor cache*. The cluster cache contains the identities of those other Yentas which share interests with this Yenta's user—in other words, if two Yentas are in each others' cluster caches, it is because each of them has at least one granule that matches a granule in the other closely enough.

The rumor cache, on the other hand, contains the identities (and representative granule text!) from the last n Yentas encountered on the net for which the local Yenta does *not* share an interest. When two Yentas communicate, they execute a referral algorithm in which Yenta A will ask Yenta B for that member of B's rumor cache which most closely matches some granule from A. Say that Yenta C is this match. Yenta A will then directly communicate with Yenta C and repeat the referral; this leads to a sort of hill-climbing approach into a collection of Yentas which are all in each other's cluster caches (because they share an interest in common). It is the need to get referrals which leads to the requirement that some sort of *content* be associated with Yentas in the rumor cache, as well as their identities.

For many more details on bootstrapping issues, the way that granules are formed, the way the clustering algorithm works in detail, and the performance of the overall algorithm, see [7].

The Threat Model: What Attacks May We Expect?

Given the system above, there are a wide variety of potential attacks which may be mounted by malicious or curious third parties. They generally break down as follows into *passive* attacks, in which communications are merely monitored, and *active* attacks, in which communications or the underlying agents themselves are subverted, via deletion, modification, or addition of data to the network.

Passive attacks. The most obvious attack on the Yenta system is simple monitoring of packet data; such an attack is often accomplished with a *packet sniffer*, which simply records all packets transmitted between any number of sources. If such data includes users' mail messages or files, then two Yentas which are trading this information back and forth will leak information to an *eavesdropper*.

Even if the actual communications between Yentas are perfectly encrypted, however, passive attacks can still be quite powerful. The easiest such attack, in the face of encrypted communications, is *traffic analysis*, in which the eavesdropper monitors the *pattern* of packet exchange between Yentas, even if the actual *contents* of the packets are mystery. This can be surprisingly effective: It was traffic analysis that alerted a pizza delivery service local to the Pentagon—and thus the media—when the United States was preparing a military action at the beginning of the Gulf War; when late-night deliveries of pizza suddenly jumped, it became obvious that something was up.

Active attacks. Active attacks involve disrupting the communications paths between agents, or attacking the underlying infrastructure. The most common such attack is a *spoofing* attack, in which one agent impersonates another, or some outside attacker injects packets into the communication system to simulate such an outcome. Often, spoofing is accomplished via a *replay* attack, in which prior communications between two agents are simply repeated by the outsider. Even if the plaintext of the encrypted contents of the communication are not known, such attacks can succeed so long as duplicate communications are allowed and the attacker can deduce the effect of such a repeat. For instance, if it is noticed that a cash-dispensing machine will always dispense money if a particular (encrypted) packet goes by, a simple replay can spoof the machine into disgorging additional cash.

More sophisticated attacks are certainly possible. Individual running Yentas might be subverted by a third party, such that they are no longer trustworthy. Such a subverted Yenta might use encryption keys which are known to the interloper, for example. Alternately, the attacker might create his or own own agent, which looks like a Yenta to the rest of the network, but pretends to be interested in *everything*—such a Yenta might then be used to troll for people interested in particular topics, and presumably also would be modified to disgorge anything interesting to its creator.

Finally, the actual distributed Yenta might be modified by a determined attacker at the source itself—say, by subtly introducing a trojan horse into the application at its distribution point(s). This is essentially a more-distributed and more-damaging version of the subverted-agent attack above. As an example, consider all the Web pages currently extant which proclaim, "These pages are best viewed with Netscape 2.x. Download a copy!" Now imagine what would happen if the link pointed to a carefully-modified version of Netscape that always supplied the *same* session key, known to the interloper: the result would be that anyone who took the bait would be running a version of Netscape with no security whatsoever, hence leaving themselves vulnerable to, e.g., a sniffing attack on their credit card number.

Security Design Desiderata

Yenta's security architecture is cognizant of several principles which are well-known in the security and cryptographic

communities. This section discusses several of them, and demonstrates how they have motivated various decisions taken in the design.

Security through obscurity does not work. This means that any design which depends upon secrecy of the *design* is guaranteed to fail, since secrets have a way of getting out. Since Yenta is designed to be run by a large number of individuals all across the Internet, its binaries must be public, hence security through obscurity would be untenable anyway in the face of disassemblers and reverse-engineering. (In fact, Yenta's source code is *also* public, which should *increase* confidence in the resulting system; see *Yvette* below.)

Keys are the important entity to protect. In good cryptographic algorithms, it is the *keys* that are the important data. Since keys are usually a small number of bits (hundreds or perhaps thousands at most), and since new keys are often trivial to generate, protecting keys is much easier than protecting algorithms—although key management is often the hardest and weakest point of a cryptosystem. Yenta has a variety of keys and manages them carefully.

Good cryptography is hard to design and hard to verify. Most brand-new cryptographic systems turn out to have serious flaws. Only when a system has been carefully inspected by a number of people is it reasonable to trust it. This is another reason why security through obscurity is a bad idea. Yenta depends on well-established algorithms and protocols for its fundamental security, since they have been carefully characterized.

Security is a function of the entire system, not individual pieces. This means that even good cryptography and system design is worthless if it can be compromised by bribing or threatening someone. Part of the reason for Yenta's decentralized nature is to avoid having a single point of compromise.

Malevolence and poor design are sometimes indistinguishable. Many system failures that look like the result of malevolence are instead the result of the interaction of an accident and some unfortunate element of the design. For example, the entire ARPAnet failed one Sunday morning in 1973 due to a double-bit error in a single IMP [12]. Or take this quote: "The whole thing was an accident. No saboteur could have been so wildly optimistic as to think he could destroy an airplane this way," which was used to describe how an aircraft was demolished on a friendly airfield during World War II when someone ingeniously circumvented safety measures and inadvertently connected a mislabelled hydrogen cylinder to the plane's oxygen system [16].

If you don't want to be subpoenaed for it, don't collect it. Federal Express, a delivery service in the United States, receives (and hence is compelled to grant) several hundred subpoenas a *day* for its shipping records [17]. The safest way to protect private data collected from others from such disclosure—not to mention the hassle of responding to a stream of subpoenas—is never to collect it in the first place. Both the lending records of most libraries, and the logfiles of MIT's primary mailers (which are guaranteed to be thrown away irretrievably when three days old) adhere to this rule. This also motivates Yentas's decentralized design;any central point is a subpoena target.

Security is a goal, not an absolute. A computer can often be made perfectly secure by unplugging it—not to mention vaporizing its disks (and their backups...). However, this is a high price to pay. Tradeoffs between security and functionality or performance are often necessary. It is also true that new attacks are constantly being invented; hence, while this research aims at a *more-secure* implementation than that which is possible without attending to these issues at all, it can never claim to be *completely secure.* We therefore aim for security that is *good enough*, such that user privacy is protected as well or nearly as well as it would be if Yenta was not running; we cannot hope for better, and may have to make tradeoffs that nonetheless lead to a little bit of insecurity for a large benefit.

Problems *Not* Addressed

There are a number of problems which are *not* addressed in the security architecture to be presented. For instance, since each Yenta runs on a user's individual workstation, and each Yenta is not itself a mobile agent per se [2][8][9][18], we do not have the problem of executing arbitrary chunks of possibly-untrusted code on the user's local workstation.

Further, it is assumed that, while *some* Yentas may have been deliberately compromised, the vast majority of them have not. This mostly frees us from having to worry about the problems of *Byzantine failure* [4][13] in the system design, wherein a *large* portion of the participants are either malfunctioning or actively malicious.

We also assume, as in the Byzantine case, that not *every* agent any *particular* Yenta communicates with is compromised. If this were not true, certain parts of the algorithm would be vulnerable to a *ubiquitous* form of the *man-in-the-middle* attack, wherein an interloper pretends to be A while talking to B, and B while talking to A, with neither of them the wiser. (Weaker forms of this, wherein there are only a *few* agents doing this, have reasonable solutions).

In addition, we do not explicitly deal with *denial-of-service* attacks, which are extremely difficult for any distributed system to address. Such attacks amount to, for example, dropping every packet between two Yentas that are trying to communicate; this attack looks like a network partition to the Yentas involved, and there is little defense.

Finally, we have the problem of Yenta's use of strong cryptography to protect users' privacy. Since the United States government currently regulates such cryptographic software as a munition (under *ITAR*, the International Treaty On Arms Regulations [5]), the cryptographic portions of Yenta's software are currently unavailable outside the US unless added back in elsewhere. This somewhat complicates parts of its design; solving the limitations of ITAR is not explicitly addressed here.

Some Useful Cryptographic Techniques

This section introduces some useful cryptographic techniques that will be used later. For a much more complete introduction that includes an excellent survey of the field, see [14].

Symmetric Encryption

One of the most straightforward cryptographic techniques uses *symmetric keys*. Algorithms such as IDEA ([14] pp. 319-324) work this way. Given a 128-bit key, the algorithm takes *plaintext* and converts it to *ciphertext*. Given the same key, it also converts ciphertext back into plaintext. Expressed mathematically, we can say that C=K(P) [the ciphertext is computed from the plaintext via a function of the key K], and similarly P=K(C) [the reverse also works].

IDEA is probably very secure. The problem comes in *distributing the keys*: we cannot just transmit the keys before the encrypted message (after all, the channel is deemed insecure or we wouldn't need encryption in the first place!), hence users must first meet (*out-of-band*, e.g., not using the insecure channel) to exchange keys. This is clumsy.

Public-key Encryption

A better approach uses a *public-key cryptosystem* [PKC], such as RSA ([14] pp. 466-473) or the many other variants of this technology. In a public key system, each user has *two* keys: a *public* key and a *secret* key, which must be generated together—neither is useful without the other. As its name implies, each user's public key really is public—it can be published in the newspaper. The secret key, on the other hand, is *never* shared, not even with someone the user wishes to communicate with.

User A encrypts a message to B by computing $C=K_{PB}(P)$, e.g., a function involving B's *public* key. To decrypt, B computes $P=K_{SB}(C)$, e.g., B's *secret* key. Note that, once encrypted, A *cannot* decrypt the resulting message, using any key A has access to—the encryption acts *one-way* if A does not have B's secret key (and she shouldn't!). [One important detail: since PKC's are usually slow, one usually creates a brand-new *session key*, transmits *that* using PKC, then uses the session key with a symmetric cipher such as IDEA to transmit the actual message.]

This scheme provides not only *confidentiality* (third parties cannot read the messages), but also *authenticity* (B can prove that A sent the message). How does this work? Before A sends a message, she first *signs* the message by encrypting it (really a *cryptographic hash* of the message—see below) with *her own private key*. In other words, A computes $P_{signed}=K_{SA}(P)$. Then, A *encrypts* the message to B, computing $C=K_{PB}(P_{signed})$. B, upon receiving the message, computes $P_{signed}=K_{SB}(C)$, which recovers the plaintext, and can then *verify* A's signature by computing $P=K_{PA}(P_{signed})$. B can do this, because he is using A's *public* key to make the computation; on the other hand, for this to have worked at all, A must have sent it, because only her *secret* key could have signed such that her public key worked to check it. Only if someone had cracked or stolen A's secret key could the signature have been forged.

Cryptographic Hashes

It is often the case that one merely wishes to know whether some message has been tampered with, without having to transmit a copy out of band. One easy way to do this is via a cryptographic hash, such as MD5 ([14], pp. 436-441) or the Secure Hash Algorithm (SHA, [14], pp. 442-445). These hash functions compute a short (128-bit or 160-bit, respectively) *message digest* of an unlimited-length original message, with the unusual property that changing *any single bit* of the original message changes, on average, *half* of the bits of the digest, in a one-way fashion—it is infeasible, given a digest, to compute a message which, when hashed, would yield the given digest. On the other hand, *anyone* can compute the hash of a message, since the algorithm is public and uses no keys.

Since such hashes are compact yet give an unambigous indication of whether the original message has been altered, they are often used to implement *digital signatures* such as in the RSA scheme above—what is signed is not the actual cleartext message, but a hash of it. This also improves the speed of signing (since signing a 128- or 160-bit hash is much faster than signing a long message), and the actual security of the cipher as well (because RSA is vulnerable to a *chosen-plaintext* attack; see [14], p. 471).

Key distribution. One of the hardest problems of most cryptosystems, even public-key systems, is correctly *distributing* and *managing* keys. In a public-key system, the obvious attacks (compromise of the actual secret key) are often relatively easy to guard against (keep the secret key in memory as little as possible, encrypt it on disk using DES with a passphrase typed in by the user to unlock it [19], and keep it offline on a floppy if possible).

But consider this: Alice wishes to send a message to Bob. She looks up Bob's public key, but interloper Mallot intercedes and supplies *his own* public key. Alice has no way of knowing that Mallot has done so, but the result of her encryption is a message that *only* Mallot, and not Bob, can read! Even if one demands that Alice and Bob have a round-trip conversation to prove that they can communicate, Mallot could be playing man-in-the-middle, simultaneously decrypting and re-encrypting in both directions as appropriate.

Systems such as Privacy Enhanced Mail [11] use a centralized, tree-structured key registry, which is inconsistent with Yenta's decentralized goals. On the other hand, PGP [19] functions with completely decentralized keys, by having users *sign each other's keys*. When Alice gets "Bob's" public key, she checks its signatures to see if *someone she trusts* has signed that key, or some short chain of trustable people, etc. If so, then this key must be genuine (or there is a conspiracy afoot amongst their mutual friends); if not, then the key may be a forgery. This practice of signing the keys of those you vouch for is called the *PGP Web of Trust* and is the primary safeguard against forged keys.

Structure of the Solutions

This section presents some solutions to some likely security problems in Yenta, using some of the technology mentioned previously. It presents a *range* of solutions; not every user might want the overhead of the most complete protection, and the elements, while often solving separate problems, sometimes also act synergistically to improve the situation. Finally, for brevity, it omits many details present in the complete design.

The Nature of Identity

Uniqueness and confidentiality. It should not be possible to easily spoof the identity of a Yenta, for a number of reasons (one major reason is discussed immediately below), yet *anonymity* is very important for controlling the impact of information disclosure and to reduce the risk of introductions. For this reason, every Yenta sports a unique *cryptographic identity*—a *digital pseudonym*. This identity corresponds, essentially, to the *key fingerprint* [19] of the individual Yenta's public key—a short (128 bits) cryptographic hash of the entire key. In order to keep some interloper from stealing, say, Yenta A's identity, any Yenta communicating with A encrypts messages using A's public key. A can prove that its identity is genuine by being able to decrypt; further, such communications have an internal sequence number (itself encrypted) to prevent replay attacks by a man in the middle. Further, of course, such encryption prevents an eavesdropper from intercepting the actual conversation.

The completely decentralized nature of Yenta complicates key distribution. The model adopted is the decentralized model used by PGP [19]. By not relying on a central registry, we eliminate that particular class of failures. And interestingly, the Yenta architecture partially eliminates the disadvantage of PGP's decentralized key distribution—that of guaranteeing that any particular public key really does correspond to the individidual for which it is claimed. In PGP, we care strongly about actual individuals, but in Yenta, only the cryptographic ID's are important—indeed, Yenta tries to *hide* the true identity of its users unless they arrange (via an introduction) to be known to each other.

Spamming and spoofing. Unfortunately, this pseudonymity comes at a price: When an introduction is about to be made, how can we have any idea who we might be about to be introduced to? Can we know that the last 10 Yentas we've seen do not all surreptitiously belong the same individual? Can we know that this person won't spam us with junk mail once he discovers our interest in a particular topic? And so forth.

Yenta solves this via a system of *attestations*. Any given Yenta may have any number of text strings associated with it, chosen by its user and signed by any number of other Yentas. Such strings might be "I am not a junk-mailer" or "I really am male" or whatever else the user wishes to write. When an introduction is to be made, both parties can check each other's attestations. If some important attestation is not signed by a sufficient number of possibly-trustable people to whom the user has already been introduced (and whose identities might themselves be checked via the web of trust), it might be treated with suspicion, and perhaps the introduction aborted. Note that the ultimate decision must be made by a human, since it is human trust in the quality of signatures we are talking about.

Additionally, we can use a distributed, linked, timestamping protocol ([14], p. 77) to make strong assertions about *how long this key has been around.* Such a protocol can allow us to trust that, at the very least, if the Yenta we are talking to is bogusly hiding its true identity via a faked web of attestations, it must have had at least n months of time invested in doing so. This biases our trust towards "old-timers."

Eavesdropping

The generally-encrypted nature of inter-Yenta communication makes most eavesdropping, including some but not all man-in-the-middle attacks, quite difficult. However, traffic analysis is still a possibility—if an interloper knows what one Yenta is interested in, watching who it clusters with could be useful.

Fortunately, Yenta can take advantage of the sort of *random-reforwarding* used in Cypherpunks remailer chains. In this scheme, any given message between Yentas A and B is first routed through other Yentas known in common between then (with the potential for further reroutes along the chain, so long as the endpoints are preserved). Such Yentas might be taken from either the cluster or rumor caches, though the latter is preferred if possible for maximal "spreading" of message traffic. For maximal robustness, messages should be padded, reordered, and delayed by random amounts to foil "input-and-output" monitoring along the path, a la the "ideal digital mix" [1].

Malicious Agents

If some malicious person was running a subverted version of Yenta, what could he discover? The most important information consists of the identities of other Yentas in the cluster cache (especially if those identities can be "real" identities and not digital pseudonyms) and the text in the rumor cache (especially if, again, such text can be correlated to real people). There are therefore two general strategies to combat this: hiding real identifying information as well as possible, and minimizing the amount of text stored in the rumor cache.

To accomplish the first, consider that Yenta A must somehow be able to communicate with Yenta B; that implies that, unless we broadcast all traffic to everyone, *somebody* must know the two IP addresses involved—but it need not be A and B. Instead, we can use random reforwarding to pass traffic to any number of intermediate Yentas, and *they* can use an *(n,k) secret-sharing* protocol [10] to jointly reconstruct the IP address of either party, given that any n or more of k Yentas know part of the secret, which is spread out randomly in advance. The only single Yenta to get a relevant (endpoint) IP address (after reconstruction of the shared secret) is *not* either of the endpoint Yentas, hence it cannot read the (encrypted) traffic, yet the two endpoint Yentas cannot determine where the traffic is going. Thus is "real" identity protected.

A similar digital mix can protect the contents of the text in the rumor cache, irrespective of the IP-identity protection above. This is much simpler—instead of having Yenta A store all the grains from B in its rumor cache, B *spreads them out* to many others (for example, one per paragraph) and gives only their ID's to A. When A tries to make a referral, it asks each of the other Yentas to instead compare their scattered paragraphs and report directly back to B. This *diffuses* the individual paragraphs of text across the entire population of Yentas, meaning that no one Yenta is in a position to reveal very much about the text—it takes large-scale collusion.

Protecting the Distribution

There is a final piece of the puzzle—how do users of Yenta know that their copy is trustworthy? The easiest approach, of course, is to cryptographically sign the binaries, such that any given binary may be checked for tampering with the authoritative distribution point. But what if the program itself, at the distribution point, had a trojan horse inserted into its *source*, either by the implementors themselves, or by a malicious third party who penetrates the development machine? Even though the source is freely distributed, and may be recompiled by end-users and checked against the binary, what individual user would want to read the *entire* source to check for malicious inclusions? (This is, of course, a problem for *any* software, and not just Yenta—but Yenta is a particularly difficult piece of software for a user to verify solely from its behavior, since it both reads sensitive files *and* engages in a lot of network traffic—and even worse, the traffic is encrypted, so one cannot even check up on it with a packet sniffer!)

To combat this, we have developed *Yvette*,[1] a Web-based tool which allows multiple people to *collaboratively* evaluate Yenta's source code, storing cryptographically-signed (hence traceable and non-spoofable) comments on particular pieces of the source where others can view them. Each individual need only check a small piece of the whole, yet anyone can examine the collected comments and decide whether their contents and coverage add up to an evaluation one can trust.

Yvette presents an interface, via the Web, which allows anyone to ask questions such as:

- Who has commented on this particular piece of code? Are the comments mostly favorable, or not? What is the exact text of the comment(s)?
- What regions have the most or least number of comments associated with them?

Yvette users may also take actions such as:

- Download, for inspection and comment, a piece of the source, which can be a region of lines in a file, a subroutine, a set of subroutines, a set of files, or an entire directory tree.[2]
- Upload cryptographically-signed comments about some piece of downloaded source code.

Using Yvette, therefore, users who wish to help verify a distribution can bite off a small piece of the problem, asking the Yvette server for which pieces of source code have not yet been extensively vetted, perusing other people's comments, and so forth. Users with no programming experience, but who nonetheless wish to check the distribution, may look at everyone else's comments to assure themselves of the integrity of the product.

Yvette thus attempts to encourage a *whole-system* approach to security, in which not only are the agents themselves secure, but their users (who are also part of the system) may easily *trust* the agents' security and integrity. It is hoped that mechanisms such as Yvette will become more popular in software distribution in general, and that it encourages thinking about more than just protocols and cryptography—if we expect widespread adoption of sophisticated agents, the sociology of how users can use and trust them matters, too.

1.The name comes from "the Yenta code vetter."

2.Unfortunately, since it distributes code that may include cryptographic routines whose export from the US and Canada is illegal [5], Yvette must also be aware of which sections of code are sensitive and must use address-based heuristics and questions of the user—only for those parts of Yenta which are cryptographic—to ensure that ITAR's export restrictions are not violated. The heuristics used are the same as those used to control the export of PGP [19], which, while easy to circumvent, are viewed as sufficient by the relevant portions of the US government.

Related Work

We are explicitly examining here only *security designs for multi-agent systems*—not multiple agents, security, or cryptography in general. This is a relatively young area. Most efforts to date have focussed on security architectures for KQML (e.g., [15]) and on Java [2][9] and bugs in its security [3]. There is considerable room for fruitful interaction between researchers working on agents and those with experience in security and cryptography.

Conclusions

Multi-agent systems which handle personal data pose challenging problems of privacy and security. Careful system design can mitigate most of these problems without unacceptable performace penalties.

Acknowledgments

I would like to thank my advisor, Dr. Pattie Maes, for her advice and her support of this research. This research has been supported in part by British Telecom.

References

[1] Chaum, David, "Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms," *Communications of the ACM*, volume 23 number 2, February, 1981.

[2] Cornell, Gary and Horstmann, Cay, *Core Java*, SunSoft Press, 1996.

[3] Dean, Drew, Felten, Edward, and Wallach, Dan, "Java Security: From HotJava to Netscape and Beyond," *IEEE Symposium on Security and Privacy*, Oakland, CA, May 6-8, 1996.

[4] Feldman, Paul, and Micali, Silvio, "Optimal Algorithms for Byzantine Agreement," *20th STOC*, pp.148-161, ACM, New York, 1988.

[5] *International Traffic in Arms Regulations*, 58 Federal Register 39,280 (1993) (to be codified at 22 C.F.R.§§120-128, 130).

[6] Foner, Leonard, "Clustering and Information Sharing in an Ecology of Cooperating Agents, or How to Gossip without Spilling the Beans," *Proceedings of the Conference on Computers, Freedom, and Privacy '95 Student Paper Winner*, Burlingame, CA, 1995.

[7] Foner, Leonard, "A Multi-Agent Referral System for Matchmaking," *PAAM '96 Proceedings*, London, England, 1996.

[8] General magic, *The Telescript Language Reference*, October 1995, {http://www.genmagic.com/Telescript/TDE/TDEDOCS_HTML/telescript.html}

[9] Gosling, J., and McGilton, H., *The Java Language Environment*, Sun Microsystems, May 1995. {http://java.sun.com/whitePaper/javawhitepaper_1.html}

[10] Ingemarsson, I., and Simmons, G. J., "A Protocol to Set Up Shared Secret Schemes Without the Assistance of a Mutually Trusted Party," *Advances in Cryptology—EUROCRYPT '90 Proceedings*, pp. 266-282, Springer-Verlag, 1991.

[11] Kaliski, Bert, "Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services," RFC1424, February 10, 1993, {ftp://ds.internic.net:/rfc/rfc1424.txt)

[12] McQuillan, J., "Software Checksumming in the IMP and Network Reliability," RFC528, June 20, 1973, {ftp://ds.internic.net/rfc/rfc528.txt}

[13] Pease, Marshall, Shostak, Robert, Lamport, Leslie, "Reaching Agreement in the Presence of Faults," *Journal of the ACM 27/2*, pp.228-234, 1980.

[14] Schneier, Bruce, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, second edition, John Wiley & Sons, 1996.

[15] Thirunavukkarasu, Chelliah, Fini, Tom, and Mayfield, James, "Secret Agents—A Security Architecture for the KQML Agent Communication Language," submitted to the CIKM '95 Intelligent Information Agents Workshop, Baltimore, MD, December 1995.

[16] Quist, Anton Braun, *Excuse Me, What Was That? Confused Recollections of Things That Didn't Go Exactly Right*, Dilithium Press, 1982.

[17] *The Wall Street Journal*, page B1, April 11, 1995.

[18] White, James, "Telescript Technology: Mobile Agents," *Software Agents*, Jeffrey Bradshaw, ed., AAAI/MIT Press, 1996.

[19] Zimmermann, Philip, *The Official PGP User's Guide*, MIT Press, 1995.