ssmail: Opportunistic Encryption in *sendmail*

Damian Bentley – Australian National University†
Greg Rose – QUALCOMM Australia
Tara Whalen – Communications Research Centre Canada

ABSTRACT

Much electronic mail is sent unencrypted, making it vulnerable to passive eavesdropping. We propose to protect email privacy by building encryption functionality into ESMTP mailers. Our solution, ssmail, provides fast, simple encryption for sendmail that does not require user intervention or reliance on public key infrastructure. We added a small number of steps to an ESMTP session, thereby allowing a client and server to create a secret, one-time session key used to encrypt the mail transfer session. ssmail relies on caching to reduce key generation overhead. The overhead imposed by our encryption scheme is minimal, allowing even busy mail servers to support privacy.

We placed our encryption mechanism within the mail transfer agent itself, allowing people to use privacy protection software without having to know how to run an encryption program explicitly. Furthermore, we are able to encrypt the email transmission session, protecting such information as sender and recipient identities. The speed and simplicity of ssmail make it a very useful addition to widely deployed ESMTP mailers. Our solution can also be adopted easily by other mailers, and can be extended to use other encryption algorithms.

Introduction

A great deal of email travelling over the Internet is vulnerable to eavesdropping. Eavesdropping operations can be done in bulk by well-funded adversaries such as organized crime or government agencies. While email is in transit from sender to receiver, it usually travels over a number of paths, any of which might be tapped by a passive listener. Encrypting email is the obvious solution to this threat. A properly used encryption algorithm is the most effective way to ensure privacy against the casual interceptor.

However, there are a number of factors that can make encryption an impractical solution. Some users are unaware of encryption technology, or how to use it properly. Furthermore, encryption algorithms can be computationally intensive, rendering them unfit for overburdened mail servers. Lastly, end to end encryption of email requires a pervasive key management infrastructure and, despite some progress, no such infrastructure exists today.

For these reasons, almost all email is currently sent unencrypted over the Internet. Our goal for this work is to significantly increase the proportion of electronic mail that is encrypted and protected from bulk eavesdropping. It is *not* our goal to make email transmission totally secure – this is better done at the application layer using, for example, PGP [19].

To address these issues, we implemented ssmail, a patch for sendmail[1] that allows for the encryption of mail sent using the Extended Simple Mail Transfer Protocol (ESMTP) [8]. By adding a few extra steps to an ESMTP session, we allow a client and server to exchange enough information to create a secret, one-time session key that is used to encrypt the mail transfer session. Our solution includes careful use of caching to reduce key generation overhead. By placing encryption within the mail transfer agent itself, we allow users to take advantage of privacy protection without having to run an encryption program explicitly. An added benefit is that all of the transmission session, including such things as sender and recipient identities, is encrypted.

The speed and simplicity of this solution make it a very useful addition to widely deployed ESMTP mailers.

Design goals

Our central goal when developing ssmail was to provide a fast, easy to use method to protect email from eavesdroppers. Note that a single network tap at a hub location could yield an enormous amount of information. Owing to the great deal of email that is transmitted and processed every day, any encryption method used must be fast. A complex encryption scheme, or one that requires a trusted third-party machine to distribute keys, will slow communication. In order to make encryption easy for users, we did not want to rely on public-key infrastructure or require that users know how to use encryption software properly. Rather, we wanted the encryption to be effectively transparent to users. They can determine that it

†The author worked on this project while employed with QUALCOMM Australia.

was used if they examine the mail headers; they will not, however, have to make it work themselves.

The Threat Model

Before describing ssmail in detail, it is necessary to clarify the threat model against which it defends. Email messages typically make their way from the sender to the recipient in a number of hops. The first hop is often inside an organization, such as a large company or an Internet service provider, and might be presumed to be secure. Similarly, the last hop might be made from an organization's firewall to the receiving machine. However, the hop(s) in the middle are usually across the Internet: they are insecure and thus perfect targets for mass interception.

Current implementations of ESMTP transmit all commands and email messages in the clear, making email vulnerable as it crosses many machines across a network. Our goal was to provide efficient protection of email privacy in the face of *passive* attacks. Our solution defends against passive attacks: anyone intercepting an ESMTP session that uses ssmail will be unable to discover the sender, receiver, or email contents.

However, ssmail *cannot* (and does not try to) defend against active attacks. The Diffie-Hellman key exchange used in ssmail does not authenticate the participants. An interceptor could substitute her public key for the real recipient's key, allowing her to read and possibly modify email before re-encrypting it with the real key and sending the email on its way. The sender and receiver will have no idea that their supposedly safe transmission was intercepted.

Defeating such an active attack is not an easy task. Solutions involve the use of a public key infrastructure (PKI), preferably one that uses certificates that ensure that public keys cannot be tampered with. The Station-to-Station protocol [10], for example, uses a PKI to allow the participants to verify that they received the correct keys during the Diffie-Hellman exchange. As mentioned above, there does not yet exist a pervasive key management infrastructure. When a PKI is in place, ssmail could be modified to perform the Station-to-Station protocol if authentication were deemed necessary. (Of course, when such a PKI is in place, interim solutions such as ssmail should not be necessary at all.) In this protocol, it is assumed that each party has access to the other party's public key; the ''man in the middle'' attack is foiled through the use of digital signatures. Such a protocol could offer stronger security with minimal added overhead, depending on how the PKI operates and how expensive it would be to obtain public keys securely or to send public-key certificates.

What ssmail *does* provide right now is a solution that is not risk-free, but is fast and simple, and can be deployed immediately to thwart the problem of passive listening.

Possible Solutions

There are a number of encryption methods available to email users that could guard against eavesdropping. People can use secondary programs such as PGP or S/MIME [4] to encrypt and decrypt their mail messages. The advantage of this approach is that users can select the encryption application that best serves their needs (such as speed or level of security). However, it requires that users be capable of using these applications correctly, which may be a problem for inexperienced users. A recent study indicates that this is more problematic than might be expected [17]. Furthermore, such methods do nothing to disguise the source or destination of the email, or any of the information in mail headers.

Another solution would be to rely on infrastructure that encrypts all traffic (including email) travelling over the network. IPSec and IPv6 [2] provide such a solution by supporting encryption of all IP packets. The advantages are obvious: all IP traffic is protected without the user having to use an encryption package directly. The major disadvantage is that such infrastructure is not yet widely deployed.

Our solution was to add an encryption command to ESMTP. This command provides mail agents with shared secret keys through a Diffie-Hellman exchange; these keys can then be used in encrypting email. We added an encryption scheme to sendmail that makes use of these shared keys. As described below, speedy key agreement is supported through the use of a special cache. ssmail users do not have to make any special effort to ensure that encryption works. Thus we provide fast and simple encryption support for ESMTP mailers.

Our Solution: *ssmail*

Communication between an ESMTP client and server consists of a series of commands and responses that are sent along with the email message. We added a new command to this protocol: the XCRYPT command. This command has a number of parameters that allow a client and server to create secret keys.

The input and output functions of sendmail are modified to encrypt and decrypt protocol messages sent. Below, we describe in detail how the XCRYPT command is used, how the keys are created, and how efficiency is addressed using caching.

The XCRYPT Command

We added the XCRYPT command to ESMTP to allow for the exchange of keys for email encryption. It has a number of parameters:

- **version strings**: the client and server exchange strings, which indicate the version and name of the preferred encryption algorithm(s) to be used (for example, t32_1.0)
- **public keys**: the client and server exchange public keys to be used in the Diffie-Hellman key agreement scheme

- **nonces**: both the client and server generate random integers that are hashed with the shared secret key to produce a new session key for each exchange (details below).

An ESMTP session that supports XCRYPT is shown in Table 1. As the table shows, the XCRYPT exchange requires five extra messages to be added to a normal ESMTP session. When a connection is made between two machines that have the ssmail patch installed, the exchange takes place as follows:

1. The server sends the initial introduction, which contains such information as the machine name and the version of sendmail being used.
2. The client sends an EHLO (extended hello) command along with its address. Sending EHLO (rather than SMTP's HELO) indicates that the client sendmail uses ESMTP.
3. The server responds to EHLO by providing a list of the ESMTP services that it supports, using XCRYPT to indicate encryption. It also sends the version string (V_b), which reports its available encryption algorithms.
4. The client then provides its version string (V_a), which includes the chosen encryption algorithm, along with its public key (K_a) and nonce (N_a). This provides the server with enough information to create the shared secret key used for encryption (detailed below).
5. The server sends its public key (K_b) and nonce (N_b), which the client uses to create the shared secret key.
6. If there are no problems with the exchange, the client sends XCRYPT OK. Failure is indicated by XCRYPT FAIL. Should a FAIL be sent in either direction, the session will continue, but the mail will be sent unencrypted.
7. If there are no problems on the server side, the server sends XCRYPT OK (else XCRYPT FAIL).

If the server is not capable of using the XCRYPT command, it is not mentioned in the list of supported services. Thus, the client will not respond with its XCRYPT command. Should the client be unable to support the XCRYPT command, this service (as sent by the server) is ignored, and the session continues normally, without encryption.

Creating Shared Secret Keys

Once the necessary information has been exchanged via the XCRYPT commands in ESMTP, the client and server use it to calculate shared secret keys used in encrypting the mail transfer. There are a number of components used in the calculation of the encryption keys, shown in Table 2.

The shared secret is generated using the Diffie-Hellman key agreement [3]. The client a and the server b share a strong prime p and a generator g. The prime and generator are hard-coded into ssmail – they

Client(a)		Server(b)	Notes
	←	220 (address)	1. Server introduction
EHLO (address)	→		2. Client introduction
	←	250 XCRYPT V_b	3. Server sends services it supports. Includes XCRYPT with version string.
XCRYPT V_a K_a N_a	→		4. Client sends version string, public key, nonce
	←	250 XCRYPT K_b N_b	5. Server sends public key, nonce
XCRYPT OK	→		6. Client XCRYPT verification
	←	250 XCRYPT OK	7. Server XCRYPT verification
Encryption starts now			

Table 1: An ESMTP session using XCRYPT.

Variable/Key	Public/Private	Notes
strong prime (p)	public	768-bit integer (static, shared)
generator (g)	public	value = 2 (static, shared)
private key (r)	private	randomly-chosen 768-bit integer
public key (K)	public	K = g^r (mod p)
shared secret	private	secret = K^r (mod p)
nonces (n)	public	randomly-chosen 80-bit integers

Table 2: Parameters used to generate secret session keys.

are the values generated for IPSec [13]. For the private keys (r_a, r_b) both parties select a randomly-generated 768-bit integer that must remain secret. To minimize the risks of an attacker discovering the private keys while they are useful, the keys are regenerated every two hours (this default value can be configured as required).

The client and server generate their public keys using their private key values. The client's public key is $K_a = g^{r_a} (mod\ p)$, while the server's is $K_b = g^{r_b} (mod\ p)$. These public keys are exchanged in the XCRYPT transfer described above. To generate the shared secret, the client computes ${K_b}^{r_a} (mod\ p)$, and the server computes ${K_a}^{r_b} (mod\ p)$. These two computations yield the same value, a number that must be kept private from other parties. Computing this shared secret is relatively expensive.

At this stage, email could be encrypted using the shared secret. However, for two reasons, there is another step in the process. As detailed below, ssmail is designed to avoid recalculating the shared secret unless absolutely necessary, which ensures that the encryption process is as fast as possible. We wish to be able to reuse the shared secret if the same client/server pair communicate again before either party's public key has changed. This is not a valid solution on its own. Re-using a key is an insecure practice: it increases the likelihood than an attacker could decrypt mail, particularly if a stream cipher is used in the encryption.

To avoid this problem, ssmail uses nonces (numbers used only once in the protocol) to ensure that a fresh private session key is generated for every exchange. The client and server exchange 80-bit randomly-selected integers. The shared secret and both nonces are passed through the Secure Hash Algorithm (SHA-1) [12] to make a 160-bit key. Messages sent from client to server use the first 80 bits as the key, and the last 80 bits are used as the key for the server to client messages.

The Encryption Algorithm

The 80-bit secret keys maintained by the client and server can be used in conjunction with any number of cryptographic algorithms. Originally, ssmail supported SOBER, a stream cipher developed by Greg Rose [15]. SOBER was specifically designed to support fast software encryption. This version of ssmail was never widely deployed, and so SOBER has been replaced. Instead, there are two algorithms supported:

- arrsyfor (compatible with RC4) [16]
- t32 (a faster and stronger derivative of the SOBER cipher) [14]

Other algorithms can be easily added, with the server suggesting the algorithm that best supports its needs.

Increasing Efficiency

Speed was a major concern when designing ssmail. Because ssmail encryption is taking place on mail servers that typically handle a large volume of mail, a slow cryptographic system could paralyse a server. The slowest part of our implementation is the Diffie-Hellman computation of the shared secret (that is, computing $K^r\ mod\ p$). To avoid recalculating this value for every ESMTP session, we use a cache. Every time a connection is made, each party stores the other party's public key and the shared secret in a cache. If this same machine is contacted later, the costly Diffie-Hellman calculation may not have to be performed again.

The format of the cache is as follows:

Header
Server's public and private keys
Time at which keys were generated
Body
machine 1: public key, shared secret
machine 2: public key, shared secret
…
machine n: public key, shared secret

Each time a connection is made, this cache is checked. First, the timestamp on this machine's own key pair is checked, and if it is too old (two hours by default), a new key pair is created and the cache is purged. Otherwise, if the other party's public key is found, then the shared secret is reused, saving an expensive recalculation. The shared secret is then passed through a secure hash computation with the nonces, but this calculation is fast. This makes computing a session key feasible for every connection.

If the public key is not found in the cache, then the shared secret is calculated with this new public key, then hashed to produce a session key. The public key and corresponding shared secret are stored in the cache for possible future use.

When a machine updates its private key, the shared secrets in the cache become invalid, as they were based on calculations performed with an outdated (different) key. The cache is simply cleared of all entries.

This caching method vastly increases the speed of encryption for a small network of machines that send mail mostly to each other (such as large corporate mail servers communicating with each other).

Performance

We performed two sets of tests on ssmail. Both tests evaluated sendmail 8.8.8 with ssmail extensions. The SSLeay library [18] version 0.9.0 generated random numbers to perform the Diffie-Hellman calculations and to perform the SHA-1 secure hash function.

The first tests were performed on ssmail with SOBER as the encryption algorithm. The test machine had a Pentium 100 MHz microprocessor and ran Linux 2.0.0. These results are based on sendmail receiving a plain text email of around 64 kilobytes with the key cache already primed (that is, no Diffie-Hellman computation).

Tests on normal sendmail (without ssmail) showed that 63.9% of program time was spent in the `collect()` function, which reads command input from the network socket. With ssmail installed, this increased to 66%, with total running time increasing by less than 5%. The `collect()` function performed almost half of the 128,000 calls to `crypto_fgetc()`, the function that delivers decrypted text using the SOBER algorithm. Similar results were found for sending email as well.

These results may vary for mailers other than sendmail 8.8.8, as sendmail makes extensive use of single character input and output. Other mailers with different methods of text processing, such as line-based methods, may have different overheads.

It must be noted that the Diffie-Hellman computation will slow down mail processing. This computation is performed every two hours (depending on the configuration), and whenever a new machine is contacted. On the Linux machine, this computation took approximately 0.5 seconds of CPU time.

Another set of performance measurements were undertaken after implementing the two new cipher methods (arrsyfor and t32). In the following discussion, all measurements were done on a COMPAQ DeskPro (Pentium Pro) at 233 MHz, compiled with `gcc -O3` on Linux 2.0.33.

One of the authors' mail traffic for just over two weeks was analyzed to try to estimate the effect of caching on the Diffie-Hellman computation. Over a period of 372.6 hours, 1,322 email messages were received with a mean size of 14,671 bytes (median 2,950 bytes). Of these, 994 were delivered by the corporate mail hub machine. In the following analysis, we assume the worst possible case. For example, we assume that a new Diffie-Hellman key would have to be computed every two hours. (This could occur if messages arrived steadily.) Furthermore, we assume that none of the communications with other machines (other than the corporate mail hub) were able to use the cached information. This is unlikely, but possible if all other machines sent at most one message each per two-hour cache period.

Let us consider the effect of caching on the Diffie-Hellman calculations. In the case of the corporate mail hub, one calculation is done every two hours, giving 187 key calculations over the test period. This leaves 807 messages sent efficiently using the cache. We must also consider the messages from other machines (assumed to miss the cache), a total of 328. Out of 1322 messages, 515 required the Diffie-Hellman computation, while 808 did not. Thus, the cache saved over 60% of these computations. This is a conservative estimate, and the benefit to the mail hub machine itself could be expected to be much greater.

Each Diffie-Hellman computation took 0.3387 s. Amortized over all email messages, this was an average of 0.1317 s per email message. This is a similar order of magnitude to the total processing time for an average message. Using lightweight Diffie-Hellman key agreement and an aggressive caching strategy adds little overhead to mail processing.

Once the shared secret has been calculated or retrieved from the cache, the session keys must be derived, the encryption algorithm must be keyed independently for each direction, and the traffic must be encrypted. Ignoring the command dialog, the bulk of the encryption applies to the email message contents. Table 3 shows the CPU time in μs taken for the various steps of each of the currently supported encryption algorithms. (Note that t32 is still undergoing changes to make it more secure, and thus its performance is likely to change slightly in the near future.)

SOBER, as used in the earlier round of testing, was somewhat less efficient than arrsyfor, so the total running time of sendmail with the new version of ssmail is expected to increase by less than 3% (not counting the Diffie-Hellman computation).

Operation	arrsyfor	t32
key generation (SHA1)	22.12	22.12
two key setups	127.6	6.06
message encryption arrsyfor speed: 8.947 MB/s t32 speed: 12.09 MB/s	 1640 	 1213
total	1790	1241

Table 3: CPU time taken.

Related work

There have been a number of projects developed to protect email from eavesdroppers. Our own work was inspired by John Gilmore's Secure Wide Area Network (S/WAN) project [5]. S/WAN is designed to protect IP traffic by adding IPSec protocol support to personal computers. This differs from our work by providing network-layer encryption that can protect more kinds of traffic than just email transmissions. However, ssmail offers a simple and fast solution for email protection that can be easily installed. This is an alternative for systems that cannot handle the overhead of IPSec protocols.

Paul Hoffman has proposed an SMTP extension for privacy and authentication using SSL [7]. If authentication is not used, then his solution is very similar to ssmail. However, Hoffman's scheme has no support for the caching of the shared secret value, which leads to greater overhead from increased public-key calculations.

There have been a number of projects that proposed to embed encryption programs in mail user agents. For example, Paul Leyland and Piete Brooks proposed a secure email project for the JANET network that would imbed PGP into mail user agents [9]. Putting encryption into the user agent rather than the transport agent requires more applications to be modified. Furthermore, they found that the PGP key servers would be unable to keep up with demand.

It is worth mentioning that there have been proposed extensions to SMTP for authentication. John Myers has put forth a system in which the client and server perform an authentication protocol exchange at the start of an SMTP session [11]. As ssmail is not intended to perform authentication, Myers' extension deals with a completely separate issue. It is possible that both SMTP extensions could be used by those systems that demand authentication and privacy and are prepared to handle the protocol overhead.

Limitations

While ssmail provides an effective solution for email privacy, it is has some limitations. Due to the multi-process nature of sendmail, the cache used to speed up key generation is stored on disk. The shared secret stored here is vulnerable to being retrieved and used to decrypt future email transmissions. Should this risk be considered too great, a trade-off in efficiency could be made. The disk cache can be turned off, resulting in a new shared secret for every exchange (but with the cost of the Diffie-Hellman computation). Note, however, that the email itself would also be exposed while stored on the compromised machine. We class this as an active attack, and hence outside the scope of our work.

Because SMTP transfers usually follow similar patterns, certain bytes generated by the encryption algorithm can be determined. For example, a "MAIL From:" command is followed by a "RCPT To:" command. Knowing this plaintext could provide enough information to allow an eavesdropper to determine such facts as the length of the recipient's email address. Any good encryption algorithm should minimize the information leaked from an encrypted session. However, this weakness is worth attention to ensure that the algorithm chosen does not fall prey to this threat.

Another limitation comes from the nature of store-and-forward transfer of email. Mail messages may pass through many servers en route to their destination. ssmail decrypts mail at each server before re-encrypting it for forwarding, so the mail remains unencrypted in the queue. As mail could be stored for an arbitrarily long time, this is a point of vulnerability. One way to reduce this threat is to use another program to encrypt mail during storage in the queue.

Future work

At the time of writing, ssmail is in a beta test version. There are a number of features that we intend to add or improve. We would like to replace the SSLeay random number generator with Peter Gutmann's package [6], which appears to be stronger. ssmail has already been extended to support t32 and RC4, an experience that indicates that porting other stream cipher algorithms should prove straightforward. Lastly, further analysis of traffic characteristics from a major mail hub would prove valuable for better estimating the saving from key caching.

Conclusions

We designed and implemented a fast, non-intrusive method for protecting large quantities of email against passive eavesdropping. We have supplied ESMTP with a key exchange command and added encryption algorithms to sendmail.

Our method has three main advantages:

- ssmail gives users privacy without requiring them to know how to use encryption packages correctly
- ssmail does not add excessive delay to email transmissions
- ssmail does not require widespread deployment of public key infrastructure

We believe that ssmail is a valuable tool for providing email privacy to a great number of users.

Availability

Contact Greg Rose <ggr@qualcomm.com> for information on availability of ssmail. We have implemented it as a patch to sendmail in order to prevent sendmail itself from being restricted by export regulations. At the time of writing, export licenses have been granted for recipients in five countries including the Czech Republic.

References

[1] B. Costales and E. Allman. *sendmail (Second Edition)*. O'Reilly & Associates (Sebastopol, CA), 1997.

[2] S. Deering and R. Minden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, Internet Engineering Task Force, December 1998.

[3] W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, 22 (1976), pp. 644-654.

[4] J. Galvin, S. Murphy, S. Crocker, and N. Freed. *Secure Multiparts for MIME*. RFC 1847, Internet Engineering Task Force, October 1995.

[5] J. Gilmore. *S/WAN: Securing the Internet against wiretapping*. http:/www.toad.com/gnu/swan.html .

[6] P. Gutmann. Software generation of practically strong random numbers. In *Proceedings of the Seventh USENIX Security Symposium*, San Antonio, TX, January 1998, pp. 243-257.

[7] P. Hoffman. *SMTP Service Extension for Secure SMTP over TLS*. RFC 2487, Internet Engineering Task Force, January 1999.

[8] J. Klensin, N. Freed, M. Rose, E. Stefferud and D. Crocker. *SMTP Service Extensions*. RFC

1869, Internet Engineering Task Force, November 1995.

[9] P. Leyland and P. Brooks. *Report of the UKERNA Secure Email Project*. http://www.cam.ac.uk.pgp.net/pgpnet/secemail/q4

[10] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press (Boca Raton, FL), 1997.

[11] J. Myers. *SMTP Service Extension for Authentication*. Internet Draft draft-myers-smtp-auth-12.txt, work in progress, Internet Engineering Task Force, November 1998.

[12] National Institute of Standards and Technology, NIST FIPS PUB 180-1, "Secure Hash Standard," U.S. Department of Commerce, April 1997.

[13] H. Orman. *The OAKLEY key determination protocol*. RFC 2412, Internet Engineering Task Force, November 1998.

[14] G. Rose, P. Hawkes. The T-class of SOBER Stream Ciphers. Unpublished manuscript. http://www.home.aone.net.au/ qualcomm

[15] G. Rose. A stream cipher based on linear feedback over GF(2^8). In C. Boyd and E. Dawson, eds., ACISP'98 (Lecture Notes in Computer Science 1438), Springer Verlag, 1998.

[16] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C (Second Edition)*. John Wiley and Sons (New York, NY), 1996.

[17] A. Whitten and J. D. Tyger. *Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0*. In Proceedings of the Eighth USENIX Security Symposium, Washington, D.C., August 1999, pp. 169-183.

[18] E. Young. SSLeay Libraries Version 0.9.0. http://www.cryptsoft.com/ssleay/faq.html .

[19] P. Zimmerman. *The Official PGP User's Guide*. MIT Press (Cambridge, MA), 1995.

Author Information

Damian Bentley is currently studying self-modifying computer viruses for honours at the Australian National University. Previously Damian has worked for CSIRO and DSTO (Australian Scientific/Defence research organizations), and for QUALCOMM Australia on cryptography. More details about Damian and his work can be found at http://mehta.anu.edu.au/~bendchon , or he can be emailed at <bendchon@mehta.anu.edu.au> .

Greg Rose joined QUALCOMM in July 1996 as a senior staff engineer and manager, working on cryptography and authentication for the GlobalStar satellite communication project and the CDMA cellular phone system, and to set up the office of QUALCOMM Australia. He is enjoying work immensely. Greg is also Vice President of USENIX. He can be contacted by email at <ggr@qualcomm.com>, and his personal web page is http://people.qualcomm.com/ggr .

Tara Whalen is a researcher investigating network security at the Communications Research Centre Canada. Other research interests include wireless and mobile computing, human-computer interfaces, and social implications of computing. Tara holds an MMath in Computer Science from the University of Waterloo. She can be contacted at <tara.whalen@crc.ca> .