A Secure Message Broadcast System (SMBS)

Mark Crosbie Ivan Krsul Steve Lodin Eugene H. Spafford*

The *COAST* Project
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907–1398
{mcrosbie,krsul,swlodin, spaf}@cs.purdue.edu

Technical Report CSD-TR-96-019

April 16, 1997

1 Introduction

This paper describes the design and implementation of a secure message broadcast system (SMBS). It is a secure, multi-party chat program that ensures privacy in communication and does not rely on shared secret keys. The system was built as a study of the feasibility of building effective communication tools using zero knowledge proofs.

There is a general consensus in the computer security comunity that traditional password based authentication mechanisms are insuficient in today's globally connected environment. Mechanisms such as one-time-passwords are a partial solution to the problem. The issue that these protocols don't address is the lack of mutual authentication. The Kerberos family of systems addresses the issue of mutual authentication but relies heavily on the physical security of the server and safekeeping of the password database.

The design of the SMBS system addresses the following issues:

Secure communication Message broadcasts between users will be encrypted to ensure privacy of communication.

User authentication A zero-knowledge proof based on the Amos Fiat and Adi Shamir [FS86] system will be used to authenticate users without revealing to potential eavesdroppers any information that might compromise the system. Although much has been written about zero knowledge proofs, and to the best of our knowledge, few products have actual implementations that depend on this class of authentication mechanisms, and none use them as effective authentication techniques for widespread dynamic communication protocols [FS86, LL95, GKG92, BG92].

Portable implementation The system was designed to be portable. This was achieved by using the PERL programming language [WS92] wherever possible.

There are three components to the system:

[*] Contact person for questions concerning the paper.

1. An **Authentication Server** verifies the identity of all parties in the system. Authentication is the process of validating that an entity actually is who it claims to be. An entity is not only a user, but other components of the SMBS system.

2. A **Chat Server** accepts messages from all clients and distributes them to all the other participating clients. It is the central message distribution point, and will encrypt all messages to guarantee privacy.

3. A **Client** is a user who wishes to participate in a message exchange with other authenticated users. A user interacts with a client by typing messages to be displayed on other user's screens, and by reading messages typed by other participating users. The client program authenticates itself to the authentication server before the user is allowed to join the system.

We will present detailed state-machines for the behaviour of each of the components of the system later in this document, but the steps involved can be summarized below:

1. The chat server and the authentication server authenticate each other using a zero-knowledge proof [GMW86].
2. Users authenticate themselves to the authentication server and obtain from it a session key. The session key is used to encrypt some of the traffic to and from the user. Immediately after this key has been obtained, users obtain from the chat server a broadcast key that will be used to decrypt broadcast messages.
3. The chat server listens on a well known port for connections. Connections will come from either users or the authentication server.
4. The chat server learns about new users when the authentication server notifies it of valid users. The chat server adds the new user to an internal database of registered users, and remembers the user's session key.
5. When a user connects, the chat server verifies that the incoming message has been encrypted using the correct session key for that user.
6. The chat server broadcasts the message to all registered users, encrypting it with a global broadcast key. It is encrypted with one broadcast key, rather than each individual user's key, to speed up distributing a message.
7. Any anomaly in the communication between the chat server and the client results in the revocation of the session key for the client, the generation of a new broadcast key by the chat server and the redistribution of the key to the remaining clients. This could occur if the client does not acknowledge the receipt of a message. The chat server will assume that the client has died, and will remove that client from its database of registered users. The client will have to re-authenticate itself with the authentication server.
8. The authentication server continues to accept connections from new clients and authenticate them.

Figure 1 illustrates the three components in the system.

2 Detailed Specifications

The clients and servers in our design conform to the following specifications:

2.1 IPC Specifications

The authentication server, chat server, and clients communicate using the TCP/IP protocol. The servers operate on well known hosts and well known ports, and it is assumed that clients will know these in advanced.

Clients and servers can communicate with each other using messages. Messages are divided into a control portion and a data portion, and these can be encrypted if necessary. Figure 2 shows the structure of a message.

The message identification field (MID) specifies the originator of the message. Its length is variable, limited to 255 characters, and is specified as the first byte of the string. The only predefined values of this field are the

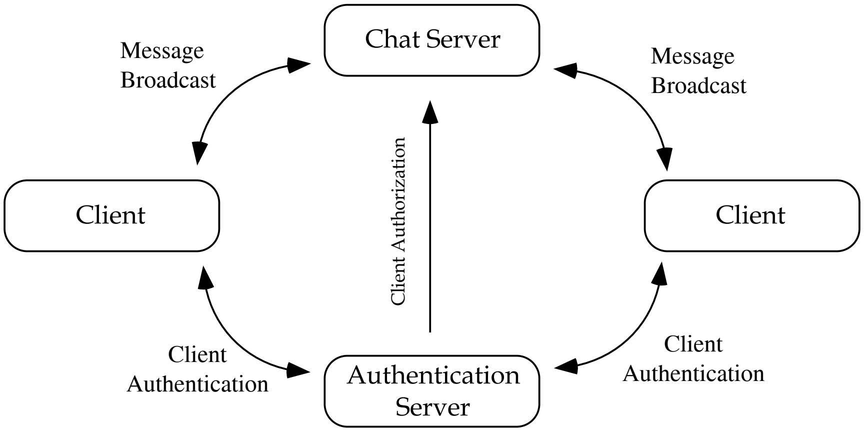

Figure 1: Conceptual Organization of the System. This diagram illustrates the types of information that can be exchanged between components in the SMBS system.

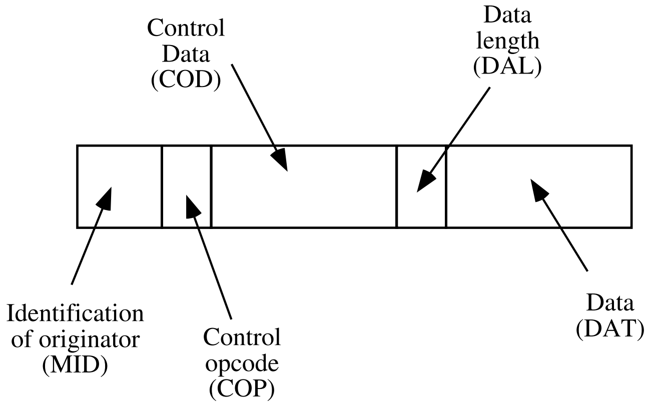

Figure 2: SMBS Message Structure. Each message passed between the components of the SMBS system is structured as shown in this figure.

OPCODE	Value	Description	Control Data Needed
ACK	0	Plain text positive acknowledgment.	None
NACK	1	Plain text negative acknowledgment.	None
EACK	2	Encrypted positive acknowledgment.	Encrypted magic cookie
ENACK	3	Encrypted negative acknowledgment.	Encrypted magic cookie
ECHAT	4	Used by clients for submitting a message to the chat server.	Encrypted magic cookie
EBCAST	5	Used by the chat server to label a message that is being broadcast by the server.	Encrypted magic cookie
CHKEY	6	Used by the chat server to inform clients that a new session key is being distributed.	Encrypted magic cookie
NEWUSR	7	Used by the authentication server to inform the chat server that a new user has just authenticated itself.	Encrypted magic cookie
DELUSR	8	Used by the authentication server to inform the chat server that a user has been revocated all authorizations.	Encrypted magic cookie
DATA	9	Plain data. Used to communicate arbitrary data.	None

Table 1: Defined values for the COD and COP fields in SMBS messages

strings[1] "authserv" and "chatserv", identifying those messages that originate at the authentication and chat servers.

The control opcode field (COP) is a one byte field that indicates the type of message. The control data field (COD) is any additional data that may be needed while specifying the type of message. The values defined for these fields are shown in Table 1.

The encrypted magic cookie is a string with the text "magic cookie ok", encrypted with the user key. The length of the encrypted magic cookie is indicated by prepending to it a four byte long length field (in network byte order)

The data length field (DAL) is four bytes long (in network byte order) and indicates the length of the payload (DAT field). The messages that carry data in the payload are shown in Table 2.

2.2 Client

On startup, the client must obtain from the authentication server a session key S_{ki} that will be used to encrypt outgoing communication with the chat server. The client does this by connecting to the authentication server's well known host and port, proving its identity using a zero knowledge proof, and obtaining the session key using the Diffie-Hellman key exchange protocol. Immediately after obtaining a session key from the authentication server, and before doing anything else, the client must wait for the chat server to send a key D_k that must be used to decrypt all incoming broadcast messages.

Once these two keys have been obtained, the client can send and receive messages. To broadcast a message the client must connect to the chat server's well known host and port and transmit a message encrypted with the S_{ki} key. To receive a message, the client accepts a connection from the server and receives a message encrypted with the D_k key.

[1] The length for these strings is not shown for readability. The implementation would have to use the strings "\0x08authserv" and "\0x07chatserv"

OPCODE	Description
ECHAT	The message submitted by the client. This message should be encrypted with the client's session key.
EBCAST	The message being broadcast by the server. This message should be encrypted with the chat server's key.
CHKEY	The new message decryption key being issued by the chat server. This message should be encrypted with the client's session key.
NEWUSR	The identification (MID) and session key for the new user. This message should be encrypted with the chat server - authentication server private session key.
DELUSR	The identification (MID). This message should be encrypted with the chat server - authentication server private session key.
DATA	Arbitrary data.

Table 2: Description of what type of messages are carried in the payload of an SMBS message

Periodically, the client will send to the chat server a keep-alive message that will contain a zero-length message[2].

To see why the keep-alive message is necessary, consider the case where the client successfully obtains the keys described above and listens without ever transmitting information (i.e. the client is passive). Assume also that the communication link between the chat server and the client is lost shortly thereafter, and for a period long enough for the chat server to try to send it some message. The chat server will assume that the client has gone away and delete it from its database of active clients. Now assume that the communication link is restored shortly after the client has been deleted. If the keep-alive message is not sent then the client would never discover that it has been deleted from the database[3].

Figure 3 shows the client's state diagram, and the following pseudo code gives a general idea of the internal structure of the actual implementation.

```
• Initialize client
• Authentication with Server
    • Perform the Zero knowledge proof and authenticate client
    • Negotiate a session key $S_{ki}$
• Wait for decrypting key from chat server
    • Save the contents of the payload as the $D_k$ key.
• Wait for connections from the network, time-outs or data from the keyboard. Loop forever
    • If we have data from the keyboard
        • Read data from the keyboard into a buffer
        • Construct a new message.
        • Encrypt the message using $S_{ki}$
        • Send message
        • Read response from the server. If we can't read or read EOF then assume that a NACK
    • If we have data from the network
        • Read and verify message
        • Decrypt message with $D_k$ key. Verify that magic cookies match.
        • If everything ok, send ACK (encrypt with key $S_{ki}$)
    • If we time-out
```

[2] Note that a zero length message will contain some data in the payload section (DAL and DAT fields) because the message must be concatenated to the magic cookie generated for the COD field.

[3] Functionally, the message is a discover-if-I-am-dead message rather than a keep-alive message

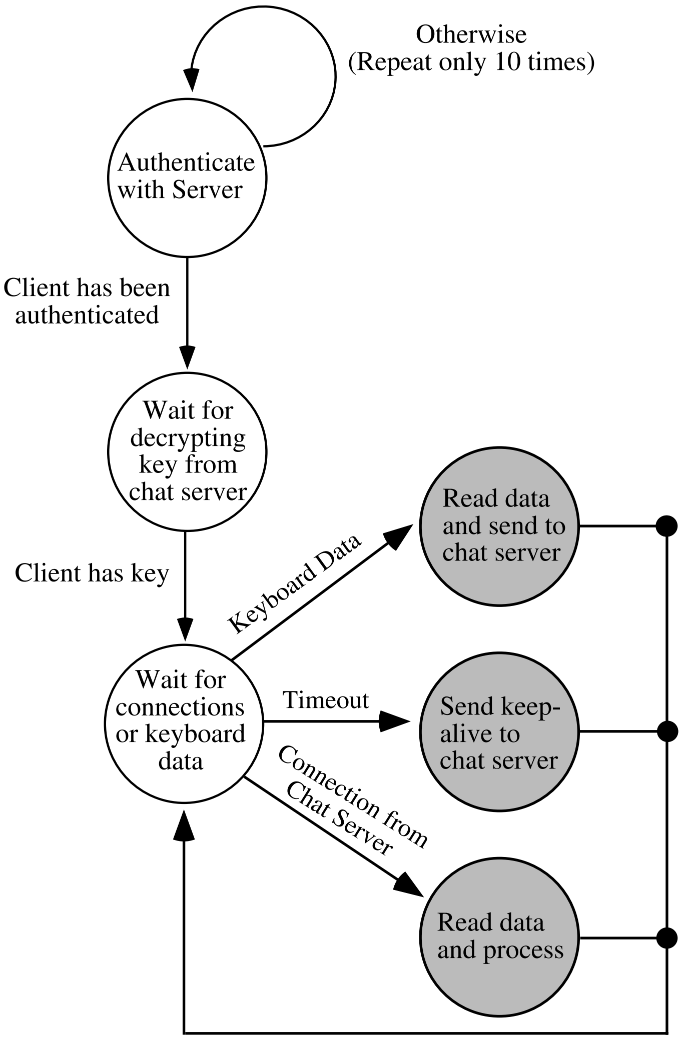
Otherwise
(Repeat only 10 times)
Authenticate with Server
Client has been authenticated
Wait for decrypting key from chat server
Client has key
Wait for connections or keyboard data
Keyboard Data
Timeout
Connection from Chat Server
Read data and send to chat server
Send keep-alive to chat server
Read data and process

Figure 3: Client State Diagram. This diagram shows the major logical steps that must be performed during the execution of a client.

- Perform the exact steps as sending a message but with an empty message.

2.3 Chat Server

The Chat server accepts incoming client connections and broadcasts data to clients for display. Each client communicates with the server by encrypting its data with a session key. This key is obtained by the client from the Authentication server at startup.

The chat server maintains a database of clients which have connected. This database contains the client's ID and session key D_k and IP address/port. It is assumed that this database is secure (i.e. no entries can be inserted by any other process other than the chat server).

The chat server has a current global key which is used to encrypt messages to be broadcast to the clients. It chooses this key at startup, and will send the key to clients as they connect. This key can be timed out and revoked as necessary.

The following pseudocode outlines the execution of the chat server:

- Chat server starts up. It initializes everything internally. The database is cleared.
- Chat server must authenticate itself with Auth. Server
- Chat server waits for connections on a well known port. Loop forever
  - Chat server receives a connection.
  - If the MID == authserv, then it is a connection from the authentication server
    - If the message type is NEWUSR, then a new user is being added to the system. In this case, add the client to the database and send him the D_k session key.
  - If the message type is DELUSR then we must delete the user in the database, generate a new D_k key and re-distribute this key to the rest of the clients
    - For each client in the database do,
      - Get the client's session key S_{ki}.
      - Send the new D_k encrypted with S_{ki}.
  - If the MID == a client ID
    - Check that message type is EBCHAT. If not, discard the message
    - If DAL == 0, ignore message, it is a keep-alive from a client.
    - Find session key S_{ki} for user. Decrypt message with session key
    - Send the message to all other clients.
      - Encrypt the message with the D_k key - current global broadcast key.
      - For each client in the database, send message to client

Figure 4 shows the Chat Server State Diagram.

2.4 Authentication Server

The SMBS authentication server is seeded through a secure method with a list of all authorized client tuples. The tuples consist of the pair (identity, large number). The authentication server's only job is to authenticate the SMBS clients and chat server using a zero knowledge proof.

The authentication server, when started, waits for incoming connections on a well-known port. Once a connection is established, the authentication server performs a zero knowledge proof with the process on the other end of the connection. The procedure will validate the identity of the incoming process. If there is a failure either with performing the Zero Knowledge Proof or with communication, the authentication server severs the connection and waits for connections again.

Once the identity has been validated, a session key is generated and exchanged using the Diffie-Hellman key exchange protocol. The session key is denoted S_k for the chat server or S_{ki} for a client.

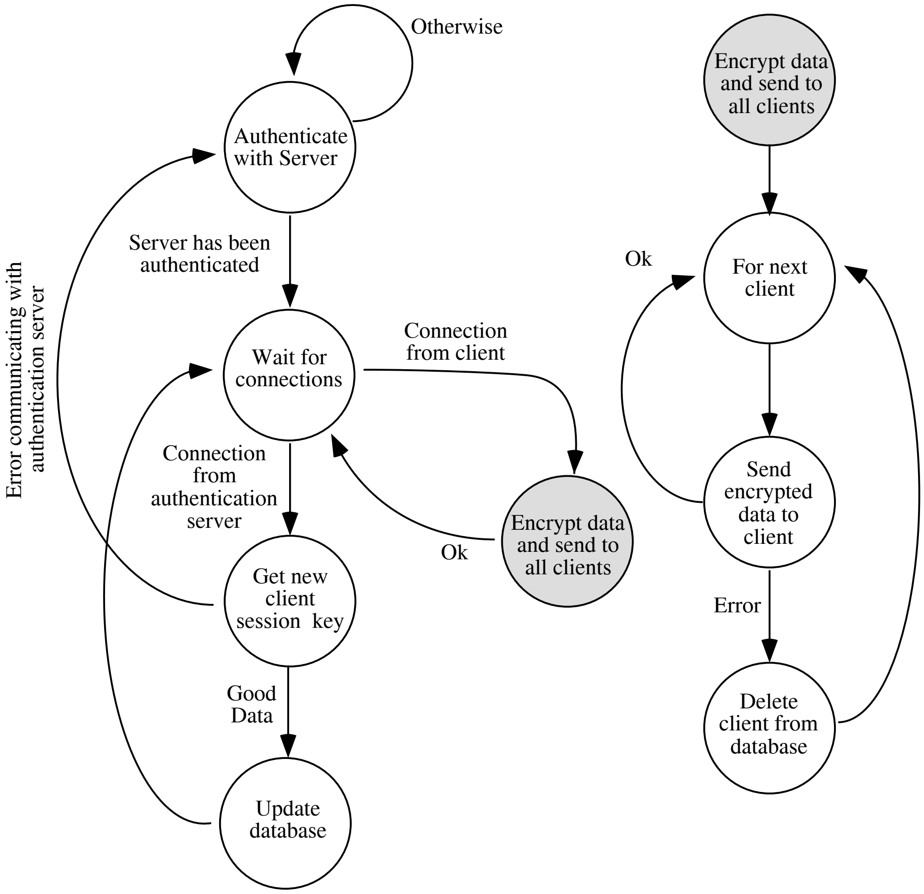
Otherwise
Authenticate with Server
Server has been authenticated
Error communicating with authentication server
Wait for connections
Connection from client
Connection from authentication server
Encrypt data and send to all clients
Ok
Get new client session key
Good Data
Update database
Encrypt data and send to all clients
For next client
Ok
Send encrypted data to client
Error
Delete client from database

Figure 4: Chat Server State Diagram. This diagram shows the major logical steps that must be performed during the execution of a chat server.

If the identity is determined to be that of the chat server, the authentication server stores the generated session key S_k and uses it for any subsequent communication with the chat server. An ACK, encrypted with the session key S_k, is sent to the chat server and the connection is closed. If either of these two communications fail, the authentication server is reset to its initial state. The authentication server then waits for incoming connections again.

If the identity of the incoming connection is determined to be a client, the authentication server first verifies that a chat server has been authenticated. If no chat server has been authenticated, the authentication server closes the connection and waits for incoming connections. If there is an authenticated chat server, the authentication server constructs a message containing the client identity and the newly generated client session key S_{ki}, encrypts the message with S_k and opens a connection to the chat server. If this connection fails, the authentication server closes the connection to the client and resets to its initial state. If the connection succeeds, it sends the encrypted message to the chat server and waits for an ACK. If no ACK is received, it closes the connection to the client and resets to its initial state. If the ACK from the chat server is received, it then ACKs the client and closes the connection to the client. The authentication server then waits for incoming connections again.

A session key S_{ki} for a client can be revoked by entering the client identification at the command line. This will cause the authentication server to generate a key revocation message and send it to the chat server.

The following pseudocode outlines the behavior of the server:

```
• No valid chat server, initial state
• Wait for connection or keyboard input loop forever. If connection established:
    • Perform Zero Knowledge Proof and generate session key using Diffie-Hellman
    • If identity is chat server
        • Store session key
        • Send encrypted ack message to chat server using session key
    • If identity is client
        • If no valid chat server - close client connection
        • Construct message with client id & client session.
        • Encrypt message with chat server session key
        • Send message to chat server
        • Send encrypted ack message to client using session key
• If keyboard input, parse the command
    • If command is revoke user
        • Parse and validate user name
        • Construct key revocation message
        • Send to chat server
    • If command is list users
        • List users registered in the database
```

Figure 5 shows the Authentication Server state diagram.

3 Cryptographic Foundations

3.1 The Zero Knowledge Proof

The prover (P) will convince the verifier (V) that he knows the prime factorization of a large composite number n, but will not reveal to V, or anyone eavesdropping on the conversation, any hint that would help him find the factors of n. We can do this because if the prime factors of n are known, then for any a one can find all

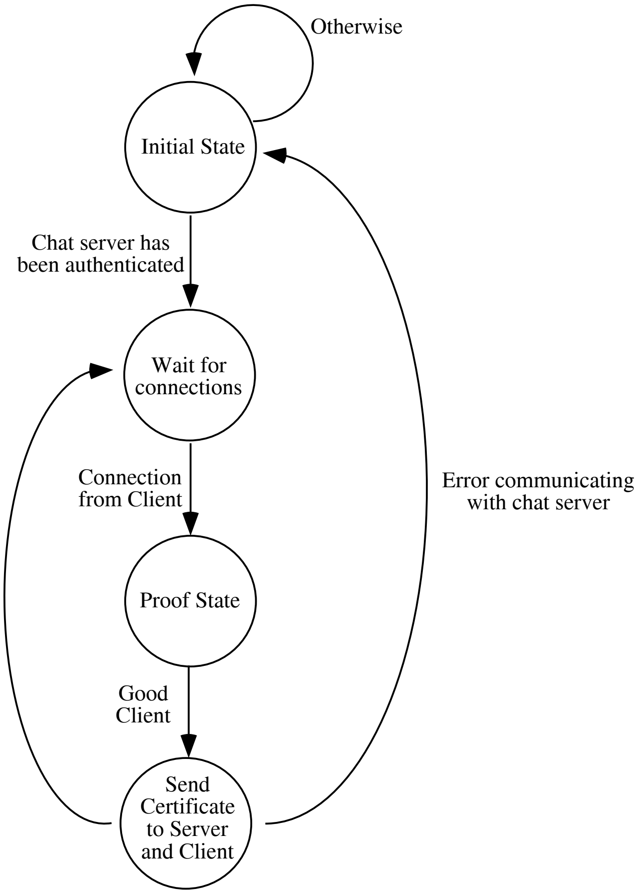
Otherwise
Initial State
Chat server has been authenticated
Wait for connections
Connection from Client
Error communicating with chat server
Proof State
Good Client
Send Certificate to Server and Client

Figure 5: Authorization Server State Diagram. This diagram shows the major logical steps that must be performed during the execution of an authorization server.

solutions of the congruence $x^2 \equiv a \pmod{n}$ in polynomial time. On the other hand, if the prime factors of n are not know then solving the congruence $x^2 \equiv a \pmod{n}$ is as hard as factoring n.

Say the prime factorization of n is $n = pq$, where p and q are primes, and assume that arithmetic $(+, -, *, \div, \mathrm{mod})$ can be done in polynomial time with large numbers. Assume also that no one knows how to factor large numbers in polynomial time.

If $n = pq$ then we can solve the congruence $x^2 \equiv a \pmod{n}$ by solving the equations $x^2 \equiv a \pmod{p}$ (yielding the solution pair $(x_1, p - x_1)$) and $x^2 \equiv a \pmod{q}$ (yielding the solution pair $(x_2, q - x_2)$). Solving equations of the form $x^2 \equiv a \pmod{p}$ can be done in the general case with the algorithm of Tonelli and Shanks [Coh93, pages 32-35]. However, for about three fourths of the prime numbers we can find the square root modulo p if we pick p to have special properties as follows:

If we pick the prime p so that $p \equiv 3 \pmod{4}$, or $p \equiv 5 \pmod{8}$, then we can find the square root modulo p with the following equations:

$$p \equiv 3 \pmod{4} \Rightarrow x = a^{(p+1)/4} \pmod{p}$$

$$p \equiv 5 \pmod{8} \Rightarrow x = \begin{cases} a^{(p+3)/8} \pmod{p} & if \quad a^{(p-1)/4} \equiv 1 \pmod{p} \\ \frac{1}{2}(4a)^{(p+3)/8} \pmod{p} & if \quad a^{(p-1)/4} \equiv -1 \pmod{p} \end{cases}$$

To find a solution to the original congruence, the solutions x_1, $p - x_1$, x_2, and $q - x_2$ are combined using the Chinese Remainder Theorem[Den83, pages 47–48] four times in the following four equations:

$t \equiv x_1 \pmod{p}$ $t \equiv x_2 \pmod{q}$	$t \equiv x_1 \pmod{p}$ $t \equiv q - x_2 \pmod{q}$
$t \equiv p - x_1 \pmod{p}$ $t \equiv x_2 \pmod{q}$	$t \equiv p - x_1 \pmod{p}$ $t \equiv q - x_2 \pmod{q}$

Four solutions to the original congruence exist because n contains two prime factors, and these four solutions will result from the application of the Chinese Remainder Theorem to these equations. Call these solutions x, $n - x$, y, and $n - y$.

The basic idea for P convincing V that he can factor n is that V presents some squares modulo n and P replies with their square roots using the mathematical properties described above. The difficulty with this simple approach is that there is a 50% chance that P will reveal to V enough information to factor n. Since V has chosen a it is reasonable to assume that V knows a solution to the square root of a modulo n. This square root must be one of the four solutions that P will compute with the above mentioned procedure. Assume then that V knows x.

If after the computation of the square root has been completed P sends x or $n - x$ to V then V learns nothing. On the other hand, if the root that P sends is y or $n - y$ then V can factor n because either $\gcd(x + y, n) = p$ or $\gcd(x + y, n) = q$.

The solution to this problem is to force V to choose an a so that he does not know any solutions to the congruence $x^2 \equiv a \pmod{n}$. This can be done by letting P choose part of a. Our algorithm requires that P and V to choose numbers b and d that will be multiplied together to produce a number $a = bd \pmod{n}$. V will not know a root to this number and hence will not be able to factor n when P returns a solution to the congruence $x^2 \equiv bd \pmod{n}$[4].

These are the steps that P and V must go through to perform the Zero Knowledge proof:

Algorithm from P's side:

[4] A word of caution. If V can trick P into solving the congruence $x^2 \equiv bd \pmod{n}$ with the same value for bd more than once then he may be able to factor n.

- Assume that P knows $n = pq$
- Choose some random number a such that $\sqrt{n} < a < n$
- Let $b = a^2 \bmod n$
- Send b to V
- Get d from V
- Solve $x^2 \equiv bd \pmod{n}$. Let x_1 be one of the four possible solutions (chosen at random).
- Get the result of a coin toss from V
- If the coin is head then send a to V. Else, send x_1 to V.

Algorithm from V's side:

- Assume that V knows n but not p, q
- Choose some random number c such that $\sqrt{n} < c < n$
- Let $d = c^2 \bmod n$
- Get b from P
- Send d to P
- Toss a fair coin so that the probability of getting heads is equal to the probability or getting tails. Send the result of that fair toss to P
- Get y from V
- If the coin toss was head verify that $y^2 \bmod n = b$, else verify that $y^2 \bmod n = bd \bmod n$.

Note that if P receives d before sending b then he could compute $b = z^2/b$ and cheat. Hence the verification step using the fair coin. Every time through the protocol P has a 50% chance of cheating V. Hence, all steps of the algorithm must be performed many times. If the results of the last step in V are always correct, after many steps V accepts that P knows the factorization of n.

After k rounds through this protocol the probability of P successfully cheating V is $\left(\frac{1}{2}\right)^k$. After 10 iterations, for example, the probability of P cheating V is one chance in 1,024. After 20 iterations the chances of P cheating V is one in 1,048,576. After 100 iterations the chances of P cheating is, approximately, one in 10^{30}. The number of iterations that must be performed then depends on the level of confidence desired.

3.2 Diffie-Hellman Key Exchange

Diffie-Hellman was the first public-key algorithm invented [Sch96, pages 513-514]. It is named after its authors [DH76]. It gets its security from the difficulty of calculating discrete logarithms in a finite field, as compared with the ease of calculating exponentiation in the same field. Diffie-Hellman can be used for key distribution. Using the familiar Alice and Bob as the participants, the algorithm is simple. First, Alice and Bob agree on two large integers, n and g, such that $1 < g < n$. These two integers do not need to be secret and they can be decided on over some insecure channel. The protocol goes as follows:

- Alice chooses a random large integer x and computes $X = g^x \bmod n$
- Bob chooses a random large integer y and computes $Y = g^y \bmod n$
- Alice sends X to Bob and Bob sends Y to Alice. Neither disclose x or y.
- Alice computes $k = Y^x \bmod n$
- Bob computes $k' = X^y \bmod n$

Both k and k' are equal to $g^{xy} \bmod n$ and they are the shared key. No one listening on the channel can compute that value since they only know n, g, X, and Y. Unless they can compute the discrete logarithm and recover x or y, they can not solve the problem.

The choice of g and n can be important to the security of the system. The modulus n should be prime and also $(n-1)/2$ should also be prime. The number g should be a primitive root mod n. One of the more important requirements is that n be large, at least 512 bits.

4 Implementation Details

We have chosen to implement the SMBS system using the Perl programming language. There are two reasons that justify this choice of programming language: 1) Perl is a highly portable language, and 2) Prototyping is much faster than C. In its current incarnation (version 5), Perl supports modular programming by way of the *Package Construct*. We made heavy use of the Perl *Package* feature. The following packages were created:

IPC Package. The IPC package provides a consistent interface to all those routines that are necessary for the construction and decoding of packets (or messages) that must be used for all interprocess communication. Among the routines provided by this package are the routines `build_ipc_message` and `decode_ipc_message`. These routines perform all the necessary encapsulation, including the creation of checksums, magic cookies, encryption of payload data, etc. Also, the message structure can be easily enhanced without affecting any other module in the system.

Because testing is an integral part of the development process we provided a comprehensive set of tests for the routines in this module in the file `test_ipc.pl`. These tests not only verify that messages are constructed correctly, but also test that the routines detect and correctly reject invalid messages or messages that have been corrupted.

ZKP (Zero Knowledge Proof) Package. The ZKP package implements the zero knowledge proof protocol as explained in section 3.1. The package provides two routines that act as the verifier (`zkp_server`) and prover (`zkp_client`). Both routines are given a file handle that is bound to the TCP/IP socket that will allow them to communicate. This file handle must allow bi-directional flow of information.

The `zkp_server` routine requires that a reference to the client database be passed as a parameter. This database is an associative array where the indices are the names of the users defined in the `keyfile` file and the values are the public keys for the clients.

The `zkp_client` routine requires that numbers p, q and n, which it needs to solve the congruences, be passed as parameters.

The `zkp_client` and `zkp_server` routines also accept an optional parameter that indicates that a visual progress feedback should be printed. The visual feedback consists of a series of asterixes (*) printed in the screen at periodic intervals.

To test the zero knowledge, a test file is provided that feeds the prover and verifier incorrect information and verifies that all errors are handled correctly. The test file will also fork off two children that will try to prove their identity to each other.

Debug Package. The package allows programs to turn on and off debugging information dynamically. Each module can define its own module identifier, and print debugging information in several levels. Level 1 is generally reserved for printing informative messages on error conditions and unusual situations that might be useful to system administrators. The debugging package allows multiple programs that share packages to turn on debugging for the same package at different levels . For example, the client program might want to display debugging information about the IPC package at level 4, but the authentication server might want to display only those messages at level 1.

MP Package. We chose to implement the bulk of the project in Perl (for portability), but unfortuantely the PERL multi-precision libraries were too slow. We used multi-precision libraries coded in C, so an interface between the two was needed. Perl allows C routines to be built in to extend the language, but this involves recompiling the local Perl installation. As we were aiming for a portable and general system, this was not acceptable.

Instead, a C program was wrapped around the MP library which accepted commands and arguments from the standard input and passed these on to the MP routines. The result from the routines were then printed to the standard output. A Perl program could then simply open a pipe to this program and pass it the required command and arguments, and read the result back in. Both commands and arguments were handled as strings. The C wrapper program then converted these into MP numbers to pass to the MP library.

A Perl package was written which encapsulated all the required MP routines. It provided a consistent interface to the MP library for other Perl code to use. It in turn called a program called `mpdispatch`. This was a C program which accepted commands (as strings) and arguments, and called the MP library. The result was then written onto the standard output, which was the end of a pipe from which the Perl program could read the result. Figure 6 illustrates this process.

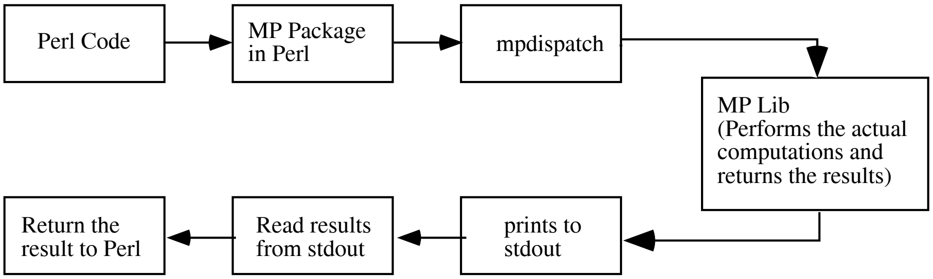
Perl Code
MP Package in Perl
mpdispatch
MP Lib (Performs the actual computations and returns the results)
prints to stdout
Read results from stdout
Return the result to Perl

Figure 6: Calling the MP library from Perl. The MP library was written in C and we use the process outlined in this diagram to interface it with SMBS (written in Perl).

The `mpdispatch` program is written in C and accepts commands and arguments from standard input. It then translates these into the internal MP format and calls the appropriate MP library routine. The result is printed on standard output.

This mechanism is flexible, but has the drawback that every mathematical operation requires a program to be executed. To offset this, we identified portions of code that did many mathematical operations, and wrote special routines to perform these operations. The Zero Knowledge Proof needed some specialized operations (e.g. solve congruence, perform Chinese Remainder Theorem). These were built into the `mpdispatch` program and increased the speed with which the ZKP could operate.

An essential part of the project was testing. As client-server systems are difficult to test, it is essential that all the lower-level modules operate correctly in situ. A test script was developed in parallel with the MP-library interface to test each routine.

A script was developed that tested the boundary conditions of each routine. Each test would print ”Passed” or ”Failed” after the test. After any modifications to the MP Perl routines, the test script was run, and all tests very verified as passing before proceeding. This script is called `testmp.pl`. Note however, it cannot test all possible inputs to the routines - we discovered problems in the MP library in taking square-roots of seemingly innocuous numbers which would never have been found with this test script.

The random number generator is needed in the Zero Knowledge Proof and Diffie-Hellman key exchange packages. Its purpose is to provide a multi-precision random number in a specified range. A requirement of this generator was that it must be quick, efficient and reasonably random. Random numbers are difficult to generate on computers, but there some well known sources of randomness in any system. We chose to use the audio device available on most Sun SPARCstations, in conjunction with the time of day.

The `randnum()` routine in PERL takes two parameters which give the upper and lower range of the random number to be generated. These are multi-precision numbers. If they are equal (i.e. there is no range) then their value is returned (e.g. `randnum(15,15)` returns 15). If the lower range is greater than the upper range, they are switched (e.g. `randnum(30,15)` is equivalent to `randnum(15,30)`).

The `randnum()` call is interpreted by the `mpdispatch()`, which routine calls the `generate_rand()` routine in C. This attempts to open the audio device `/dev/audio` and read data from it. This data is hopefully random (if the microphone is turned on), and is used to build a random multi-precision number. The time of day is added to each digit to introduce extra randomness if the microphone is not enabled. The audio device must be present for the random number generator to work. Furthermore, it must be turned on - a check is made at the start of the mp.pl package to see if the audio device is returning information which would indicate that the microphone is turned off. If so, the package exits with an error.

Crypt Package. The crypt package provides an interface to the encryption/decryption program. The encryption and decryption processes in this project are performed using the IDEA program [LM91, Lai92]. There are two routines in the crypt package, `encrypt` and `decrypt`. Both routines take the same arguments as input: a string S and a key k. The string S must be non-zero in length. Also, it must be less than 1024 characters, which is the IDEA program buffer size. The key k is a string with non-whitespace characters used to encrypt or decrypt the string S.

The data encrypted by IDEA is normally located in files and is generally binary, not ASCII format. This required the use of encoding/decoding routines to transform the binary data into some usable ASCII representation and back again. Every instance of encrypted data is encoded and manipulated in this format. Before being decrypted, this data is decoded into the native format.

Both the encrypt and decrypt routines return an array of information. The first value is the status of the operation, which can be no error, or a failure mode described below. If no error occurred, the second value is the returned string. The output from the encryption routine is the encoded encrypted string S' given by $S' = \mathit{encode}(E_k(S))$ where E is the IDEA program, k is the encryption key, and *encode* is the encoding program. The output of the decryption routine is S' given by the expression $S' = D_k(\mathit{decode}(S))$ where D is the IDEA program, k is the key, and *decode* is the decoding program.

We've chosen to use the IDEA block cipher for encryption and decryption. According to Schneier [Sch96, page 319], IDEA appears to be significantly more secure than DES and the software implementations are about as fast as DES. IDEA operates on 64-bit plaintext blocks using a 128 bit key. It uses both confusion and diffusion mixing operations from different algebraic groups to provide its security. We are using IDEA in its default operating mode, cipher block chaining.

We exercise several tests to test the crypt package. These test are designed to test all combinations of valid and invalid instances of input to the crypt package, string data and keys. These tests are performed in a serial fashion with the output from the test compared to the expected output yielding a pass/fail message. The tests are designed to exercise all the failure modes of the crypt package. These modes are: no error, no data, an IDEA error, or too much data.

Testing the string data required tests using strings of various lengths and validity. The test program uses zero length input, valid length input, overlength input, and invalid data for testing. Different keys of varying length and characters are used to test the keys. For the process of argument testing, if both arguments are not passed, the routines return an error.

Diffie-Hellman Key Exchange Package. The Diffie-Hellman [DH76] key exchange package implements the protocol as explained in Schneier [Sch96, pages 513-514]. The package provides a client routine that communicates to another instance of the client routine. The two instances of the client routine are either the authentication server and a chat client or the authentication server and the chat server. Both sides of the key exchange execute the same routine. The protocol goes as follows:

- A and B agree on two large integers n and g where $1 < g < n$ and n is a safe prime and g is a primitive root of n

- A chooses a random large integer x and computer $X = g^x \bmod n$
- B chooses a random large integer y and computer $Y = g^y \bmod n$
- A sends X to B and B sends Y to A.
- A computes $k = Y^x \bmod n$.
- B computes $k = X^y \bmod n$.

The integers n and g for our Diffie-Hellman key exchange protocol implementation were derived from Maple using the following simple algorithm:

```
with(numtheory);
seed:=7413278905432789054327890546781242345708914367854320543272679984;
n:=safeprime(seed);
g:=primroot(seed/2,n);
```

Since both n and g can be public, they are stored in the file in plaintext. The protocol does not rely on security by obscurity.

The input to the client routine is a filehandle for socket communication, the identification of the server (from this key exchange client's point of view), and the identification of the other side. The client routine performs one side of the key exchange from the perspective of A (also known as Alice). The client routine uses the MP library routines for choosing a random number x between 1 and n and calculating $X = g^x \bmod n$. This number X is sent to the other side while Y is received from the other side and then the session key $k = Y^x \bmod n$ is calculated. The output of the client routine is either the exchanged session key or undefined if an error in the input or the negotiation occurred.

The process of testing the Diffie-Hellman key exchange requires that two TCP connections are established, one acting as server and the other as client. To test a valid exchange, the server and the client compute session keys and then the client sends the computed key to the server for comparison. To test for invalid input, tests are performed for invalid file handles and blank or missing identification strings.

4.1 Clients and Server

The three programs described below constitute the SMBS system.

4.1.1 Authentication Server

The main goals of the authentication server are to authenticate a client and distribute a negotiated session key. The authentication server reads its private keys p and q from a file that has been encrypted using IDEA with a pass phrase. It also reads from a file the public keys of all the potential participants, including broadcast message clients and the chat server. The authentication server then waits listening on a well-known port for incoming connections.

When an incoming connection is made, the authentication server begins the authentication process by participating in a mutual authentication protocol using a zero knowledge proof of identity where the incoming client connection tries to prove who it is by being able to solve for the prime factors in its public key read earlier. If the client successfully authenticates itself, then the authentication server tries to authenticate itself to the client using the same process. This will ensure that both the client and the server have proof of who they are communicating with.

After a successful mutual zero knowledge proof authentication, the server and the client exchange session keys using the Diffie-Hellman key exchange protocol. If the authentication client is the chat server, the session key is stored for use with all subsequent communications with the chat server. If the authentication client is a broadcast message client, the newly generated session key is encrypted and sent to the chat server provided a chat server has already been authenticated. Once all the key exchanging and handshaking is completed, the server starts to listen for incoming client connections again.

If at any point during the authentication process or key exchange protocol an error occurs, the connection is dropped and the previous connection information for that particular client is lost. Also, if the communication process to the chat server encounters a problem, the chat server must re-authenticate itself to the authentication server and re-negotiate a session key.

Commands to the authentication server can be typed in on standard input. If the command is `help`, the list of available commands will be printed. If the command is `list`, the list of users in the database will be printed. If the command is `quit`, the open connections are closed and the program exits. If the command is `revoke`, the server asks for the identification of a user to be deleted. If the identification is valid, the users is deleted from the database. A message is sent to the chat server suggesting the particular client to remove from the list of authenticated clients. This invalidates the client for the rest of the session.

Authentication Server Usage

```
% auths.pl <options>
```

If no options are specified, it uses default hostnames and default ports for the chat server.

To change the port and host being used for the chat server, use:

```
  -ch <hostname> - hostname of the chat server
  -cp <port>     - port number for communicating with chat server
```

To change the port number that the auth server listens on, use:

```
  -ap <port>      - port number to listen for connections on
```

For example, to run the auth server verbosely on port 5004 and expected the chat server to be run on host yavin port 5005, do

```
  auths.pl -v -ap 5004 -ch yavin -cp 5005
```

4.1.2 Chat Server

The chat server was built with one simple rule — if any error occurs in communicating with a client then that client is ignored from then on. Many special cases are avoided by doing this as it simplifies the code considerably.

The chat server maintains a database of all clients it currently knows about. This is simply an array in memory giving the session key for that client, its machine address and port. This information is passed to the chat server by the authentication server when a new client has successfully passed the authentication process.

The chat server authenticates itself with the authentication server and then authenticates the authentication server. Once it has done this, it negotiates a session key with the authentication server. Finally, it generates a large random number (Dk) which will be used as a session key to encrypt outgoing broadcast messages. It then waits for connections on its well known port. If a connection comes in from the authentication server, it is either a NEWUSR command or a DELUSR command. A NEWUSR message tells the chat server that a new client has successfully authenticated. The client's session key, address and port are sent to the chat server. It adds these to its database and then attempts to send the current Dk to the new client. If it fails to do this, then it removes the client from the database.

If a client connects, it is to send a chat message to other clients. The chat server decrypts the message using the client's session key, and verifies the message integrity. It then encrypts the message with the current Dk session key before sending it to each client in turn. Again, a failure to send to a client will result in the client being deleted from the database.

The chat server has options to control on which host the authentication server will be located. This allows the distribution of the programs on multiple machines, and more importantly, allows the authentication server to run on a secure machine.

The chat server also has a -test option. This performs testing on some internal routines to ensure they do correct parameter validation. This is used to ensure some basic correctness. Testing the client-server aspect of the chat server is difficult, and is done by demonstration instead.

Chat Server Usage

```
chats.pl <options>
```

If no options are specified, it "does the right thing" and starts up on default ports that should correspond to the ports in use by the authentication server. It assumes that all the clients, and the authentication server are running on the same machine. This can be changed by the command line options.

To change the port and host being used for the authentication server use:

```
  -ap <port>     -  port for communicating with the auth server
  -ah <machine> -  machine name for communicating with the auth server
```

E.g. if the auth. server is on narnia, port 5004 do:

```
  chats.pl -ah narnia -ap 5004
```

To change the port which the chat server listens on for connections use:

```
  -cp <port> - port to listen on for connections
```

For example, to run a chat server on port 5006 do

```
   chats.pl -cp 5006
```

The chat server assumes that its key is stored in a file named `pkeys_chatserv`. This is built using the `register_client` program. A password is used to encrypt this file when it is built. This password is used by the chat server to decrypt the file to extract the key and its prime factors p and q

Finally, for peace of mind, the -test option allows you to run a variety of tests on the chat server to make sure it is behaving itself. They aren't as exhaustive as I'd like, but they're better than nothing. Be sure to run the testmp.pl script to test the MP routines, and the other test scripts as well.

4.1.3 Client

As described earlier in this document, the client was built to communicate with the chat server sending and receiving encrypted information. Upon startup, the client authenticates itself to the authentication server and authenticates the authentication server (hence preventing the impersonation of the server), negotiates a session key with the authentication server using the Diffie-Hellman protocol, and waits for connections from the chat server.

When connections do happen they can be only of type EBCAST and CHKEY. In the first case, the message is decrypted with the decryption key distributed by the chat server and in the second case the client decrypts the message with the session key negotiated with the authentication server.

The client has options to control on which host the authentication and chat servers will be located, and which ports to use.

Client Usage

```
% client.pl <options>
```

If no options are specified, it uses default host names and default ports for the authentication and chat servers. To change the port and host being used for the servers use:

```
  -ch <host name> - host name of the chat server
  -cp <port>      - port number for communicating with chat server
  -ah <host name> - host name of the auth server
  -ap <port>      - port number for communicating with auth server
```

To change the port number that the client uses:

```
  -p <port>      - port number to listen for connections on
```

For example, to start the client on port 5616 and expect the chat server to be on yavin on port 5005 and the authentication server to be on narnia on port 5005, issue the command:

```
  client.pl -p 5616 -ah narnia -ap 5004 -ch yavin -cp 5005
```

4.2 Adding Authenticated Users

The process for adding users to the system is fairly straightforward. The following procedure is used:

1. The client generates Bloom primes p and q (use safeprimes in Maple) for use as the private key.
2. The client then runs the register_client.pl program. The program asks for a user identification string, p, and q. It returns with the public key n. The private keys are encrypted with IDEA using a pass phrase and stored in a file. The client should keep this pass phrase and file secure.
3. The client sends the public key to the manager of the authentication server. The user need not worry about this operation being secure as the key (n) is public.
4. The authentication server manager, after verifying that the public key (n) actually belongs to the user, edits the file ”keyfile” to add the client. The keyfile format is ID:KEY where ID is the identification string of characters used above to generate the public key and KEY is the public key n derived above.

This would be done for the authentication server, the chat server, and all clients. The authentication server and its keyfile are required to be run on a secure machine. Furthermore, the keyfile should be kept on read-only media and edited off-line.

4.3 Finding Bloom Primes in Maple

To find Bloom primes, known as a safeprime in Maple, the following simple program can be used:

```
    with(numtheory);
    p:=safeprime(seed1);
    q:=safeprime(seed2);
```

The seeds, seed1 and seed2, should be of sufficient length (around 50 digits) of a random sort of nature (i.e. type fifty random numbers with your eyes closed). The safeprime function will return the next Bloom prime after the seed.

5 Conclusions

The goal of our project was to determine the feasability of implementing a number of crytpographic algorithms. We chose a secure communication system as a test vehicle. We aimed to design a protocol that would allow secure communication, and use this to determine how feasible it is to build reliable, real-world cryptographic systems.

5.1 Future Work

Currently the random number generator uses the audio device to obtain random background noise. This depends on the availability of such an audio device — we would like to make random number generation more portable in this respect.

Prime numbers are generated using Maple. We would like to include a prime number generator to allow the system to be installed anywhere.

The servers are not multithreaded which reduces the concurrency possible. This can easily be changed.

Currently, SMBS uses IDEA as the encryption algorithm. We will allow any algorithm to be used by providing "hooks" that another encryption package can use. For example the "encipher this data block" and "decipher this data block" hooks could be overloaded by another encryption package, such as DES.

The interface to the multi-precision (MP) library is too cumbersome to be practical. We wrote the MP interface in Perl so our system could be prototyped quickly. A better implementation would be to use other crypto libraries [Lac] or use the GNU multi-precision package.

The client, chat server and authentication server could be ported to Java [Mic95]. This would enhance the portability of the system — Java is bytecode interpreted and so is platform independent. The interface to the SMBS system would be through a Web broswer running the Java applets. This convenient interface would make SMBS far easier to use.

6 Acknowledgements

The authors wish to acknowledge Dr. Samual Wagstaff and Dr. Eugene Spafford.

References

[BG92] Mihir Bellare and Oded Goldreich. On defining proofs of knowledge. In *Proceedings of CRYPTO 92: Advances in Cryptology*, Lectures in computer science. University of California, Santa Barbara, 1992.

[Coh93] Henri Cohen. *A couse in computational algebraic number theory.* Graduate texts in mathematics. Springer-Verlag, 1993.

[Den83] Dorothy E. Denning. *Cryptography and Data Security.* Addison-Wesley, 1983.

[DH76] Whitfield Diffie and Martin Hellman. New directions in cryptography. *Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proceedings of CRYPTO 86: Advances in Cryptology*, Lectures in computer science, pages 186–194. University of California, Santa Barbara, 1986.

[GKG92] Dimitris Gritzalis, Sokratis Katsikas, and Stefanos Gritzalis. A zero knowledge probabilistic login protocol. *Computers & Security*, 11(8):733–745, December 1992.

[GMW86] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity. In *Proceedings of 27th IEEE Annual Symposium on the Foundations of Computer Science*, 1986.

[Lac] Jack Lacy. Cryptolib 1.1. Announced in sci.crypt.

[Lai92] Xuejia Lai. *On the design and security of block ciphers.* Hartung-Gorre Verlag, Konstanz, Switzerland, 1992. This is the author's Ph.D. dissertation. "Secret-key block ciphers are the subject of this work. The design and security of block ciphers, together with their application in hashing techniques, are considered. In particular, iterated block ciphers that are based on iterating a weak round function several times are considered. Four basic constructions for the round function of an iterated cipher are studied.".

[LL95] Chae Hoon Lim and Pil Joong Lee. Several practical protocols for authentication and key exchange. *Information Processing Letters*, 53(2):91–96, 1995.

[LM91] X. Lai and J. L. Massey. A proposal for a new block encryption standard. In *Advances in Cryptology — Eurocrypt '90*, pages 389–404, Berlin, 1991. Springer-Verlag.

[Mic95] Sun Microsystems. *The Java Language Specification*, release 1.0 alpha 3 edition, May 1995.

[Sch96] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, Inc, New York, NY, USA, 2nd edition, 1996.

[WS92] Larry Wall and Randal Schwartz. *Programming PERL.* O'Reilly and Associates, 1992.