

**iCheck: An Architecture for Secure Transactions in the  
Processing of Bank Checks**

by  
Joseph Figueroa  
B. S. EECS  
Massachusetts Institute of Technology, 1996

Submitted to the Department of Electrical Engineering and  
Computer Science in Partial Fulfillment of the Require-  
ments for the Degree of Master of Engineering in Electrical  
Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1997

© 1997 Joseph Figueroa. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and dis-  
tribute publicly paper and electronic copies of this document in whole  
or in part, and to grants others the right to do so.

Author .....  
Department of Electrical Engineering and Computer Science  
May 23, 1997

Certified by .....  
Dr. Nishikant Sonwalkar  
Thesis Supervisor

Accepted by .....  
Arthur C. Smith  
Chairman, Department Committee on Graduate Thesis

OCT 29 1997



# **iCheck: An Architecture for Secure Transactions in the Processing of Bank Checks**

by

Joseph Figueroa

Submitted to the  
Department of Electric Engineering and Computer Science

May 23, 1997

In partial Fulfillment of the Requirements for the Degree of Master  
of Engineering in Electrical Engineering and Computer Science

## **ABSTRACT**

*iCheck* is a system architecture that enables the banking system to increase the check processing speed and decrease the check processing cost. The *iCheck* architecture provides a secure, reliable, and widely available infrastructure for accessing the existing bank payment system over the internet. This is accomplished by designing, developing and integrating the components necessary to decrease bank check processing time and to reduce the need for human intervention in a secure form over open public networks. These components are implemented in two main modules and demonstrate the inter-operability and the infrastructure that can benefit banks by allowing them a consistent, secure, trusted way of offering faster and cheaper services. This infrastructure should also enable the offering of innovative new systems which take advantage of new developments in telecommunications technology and enhancements to the existing banking payment system.

Thesis Supervisor: Dr. Nishikant Sonwalkar  
Title: Director, Hypermedia Teaching Facility

## Acknowledgments

I express my sincere gratitude for Dr. Amar Gupta who, despite his many responsibilities, was able to help me convert my wish to write a thesis into concrete, well-planned sub-goals, help me flesh out ideas, and most of all be a great friend and mentor.

I want to thank Dr. Nishikant Sonwalkar for his time and effort in reading through various drafts and providing insightful observations.

My mother Joan Figueroa and father Nadir Figueroa, whose love and support was essential for my getting through my undergraduate and graduate programs. They provided a nurturing environment at home, guided me throughout my life, and served as role models.

Miss Jesse Middendorf whose love, encouragement, and understanding helped me through my ups and downs with unquestioning support.

I want to thank both my undergraduate research assistants, Shawn Nicholson and Shiu-chung Au, for their help and hard work. Without them, it would have been difficult to complete my research on time.

Obed Torres for his company and friendship.

# Table of Contents

<b>1</b>	Introduction.....	6
<b>2</b>	Background.....	8
2.1	Net Cash.....	8
2.2	Net Check.....	11
2.3	Millicent.....	12
2.4	Visa and Master Card's Secure Electronic Transactions.....	14
2.4.1	Cardholder Registration .....	15
2.4.2	Merchant Registration.....	17
2.4.3	Purchase Request .....	18
2.4.4	Payment Authorization .....	19
2.4.5	Payment Capture.....	20
<b>3</b>	Motivation.....	21
3.1	Current Model for Processing Bank Checks.....	21
3.2	Design Goals.....	23
<b>4</b>	iCheck Model/Design .....	25
4.1	Bank Module.....	26
4.1.1	OCR Module.....	27
4.1.2.	Check and Briefcase Object Packer Module .....	28
4.1.3.	Network/Security Module.....	30
4.1.3.1	Session Key Exchange Protocol .....	33
4.1.3.2	Certification Authority.....	39
4.2	Central Storage Facility .....	41
4.2.1	Parser.....	42
<b>5</b>	iCheck Scenario .....	45
<b>6</b>	Conclusion .....	48
6.1	How <i>iCheck</i> Succeeds to Meet Design Expectations .....	48
6.1.1	Drastic Increase in Speed of Check Processing.....	48
6.1.2	Drastic Decrease in Check Processing Cost .....	49
6.1.3	Secure Transactions .....	49
6.2	Future Work.....	49
6.3	How can Other Systems Benefit from <i>iCheck</i> .....	50
<b>Appendix A</b>	Parser .....	51
<b>Appendix B</b>	Form Based File Uploading Sample HTML Page .....	54
<b>Appendix C</b>	Certification Authority .....	55
<b>Appendix D.1</b>	Three Phase Session Key Exchange Protoco (Server).....	62
<b>Appendix D.2</b>	Three Phase Session Key Exchange Protoco (Client).....	78
<b>Bibliography</b>	.....	96

# List of Figures

Figure 1: Generic exchange with the currency server .....	9
Figure 2: Accounting Hierarchy .....	11
Figure 3: Summary of carholder registration protocol .....	17
Figure 4: Summary of purchase request .....	19
Figure 5: Complete check flow.....	22
Figure 6: Typical check route .....	23
Figure 7: Global <i>iCheck</i> view .....	26
Figure 8: Check Object .....	29
Figure 9: Briefcase Object .....	29
Figure 10: Nonce session key exchange protocol.....	34
Figure 11: Three phase session key exchange protocol.....	36
Figure 12: <i>iCheck</i> public key certificate .....	39
Figure 13: Form-based file uploading.....	42
Figure 14: Central Storage Facility parser.....	43
Figure 15: Initiate Session Key Negotiation web page.....	45
Figure 16: Summary of the <i>iCheck</i> system communication between the OCR module and the Central Storage Facility .....	47

## 1.0 Introduction

Approximately 90 billion dollars each year are spent on the processing of machine printed bank checks in the United States. Several steps in the processing of these checks are done by computers, but most of the processing cost lies in the steps that require human intervention. The most costly of these steps consists of the physical transportation of the checks and the identification of the dollar (or the unit of currency) amount for which the check has been written. Employees at bank processing centers read the amount by hand and enter the quantity into a computer, which prints the amount in MICR ink (Magnetic Ink Character Recognition) at the bottom of the check. Thereafter, the check is physically forwarded through various banks where sorting machines or MICR devices are used to aid in the processing. However, banks have not fully exploited the decrease in processing costs that network connectivity, such as the internet, can potentially provide since checks are still physically moved through the bank system.

Recently there has been a rapid expansion of the use of the internet. With more than 30 million users today, and 90 million projected to come on board in the next two years, the internet is a new way for businesses to establish computer-based resources that can be accessed by consumers as well as business partners around the world [25]. Not only are individuals using the internet, but organizations and businesses are also including internet-connectivity into their information infrastructure. Thus, the number of users and organizations reachable through this worldwide network is greatly increasing. The internet is now seen by many organizations as an efficient means to reach potential customers. However, secure methods of transactions are needed before we will see widespread commercial use of the internet.

Taking into account the growth of the internet as a business model and the fact that there is a need to reduce the cost of processing bank checks, a system is proposed that can autonomously

recognize the amount for which a check has been written and can allow local banks to electronically transfer the information on the checks between themselves and other remote banks in a secure fashion. Thus, our system can reliably and securely speed up the processing of checks and greatly decrease processing costs.

## 2.0 Background

There are currently a number of initiatives underway for the creation of secure cost-effective payment systems which attempt to take advantage of internet connectivity; yet none of these systems take full advantage of the existing financial infra-structure. Furthermore, they require users to change their money-spending paradigm since there is no paper currency or physical receipts of any kind. We will now give an overview of the salient features of these systems.

### 2.1 Net Cash

Net Cash is a framework that supports “real-time electronic payments with provision of anonymity over an unsecure network”[12]. Its infrastructure is based on currency managers that are independently managed and distributed. These currency managers provide a point of exchange between non-anonymous instruments such as electronic checks and anonymous electronic currency. If an organization wishes to set up and manage a currency server it has to obtain insurance from an insurance agency similar to the Federal Deposit and Insurance Corporation (FDIC), referred to as the Federal Insurance Corporation (FIC), to establish an authentication service. After the insurance has been obtained, the currency server sends its public key to the FIC through a secure channel. The FIC then issues the currency server a “certificate of insurance for producing and managing the currency”[12]. This certificate is then used by financial institutions and other currency servers to verify the new currency server’s authenticity.

The certificate is composed of a unique ID (*Certif\_id*) to identify the particular currency server named in the certificate, the public key of the currency server *KP* along with the date of issue and an expiration date of the certificate. This certificate is sealed with the FIC’s secret key *KS<sub>FIC</sub>*. The coin itself consists of a symbolic currency server name, a currency server IP address,

expiration date, serial number, and coin value sealed with the currency server's secret key  $KS_{CS}$ .

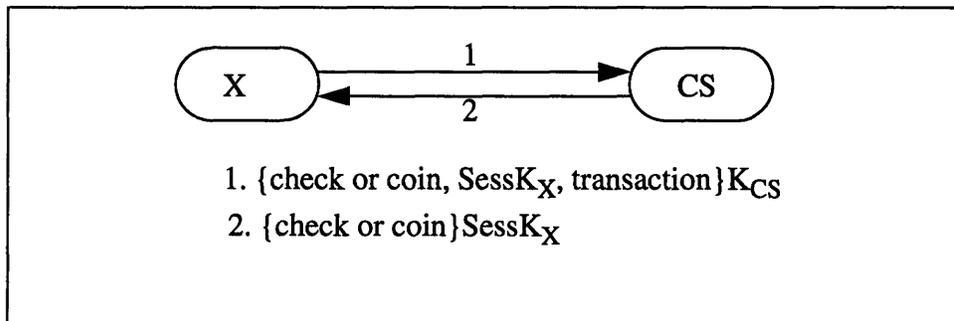
Thereafter, any coins issued by the currency server can be verified and authenticated:

**Certificate:**  $\{Certif\_id, CSname, KP_{CS}, issue\_date, exp\_date\}_{KS_{FIC}}$

**Coin:**  $\{CS\_name, s\_addr, exp\_date, serial\_number, coin\_val\}_{KS_{CS}}, Certif\_id$

A client that wants to decrypt the coin uses the *Certif\_id* to map to the appropriate certificate. After the client completes the mapping, it uses the corresponding FIC's public key  $KP_{FIC}$  to decrypt the certificate which contains the public key  $KP_{CS}$ . Then, it uses  $KP_{CS}$  to decrypt the coin. Since  $KP_{FIC}$  is FIC's public key, it can be sure that the currency server CS minted the coin.

There are four types of basic exchanges that provide the following services: coin verification (i.e., double spending detection), coin exchange for untraceability, purchasing coins with checks and cashing in coins for checks. These exchanges are showed in the figure below:



**FIGURE 1. Generic exchange with the currency server (from [12])**

Out of the four types of exchanges, only the latter two reveal the identity of X. In step 1, X sends a check or a coin encrypted with the currency server's public key  $K_{CS}$  together with the new session key  $SessK_X$ , and an indication of the transaction to be performed.

If a coin sent by  $X$  was issued by the currency server itself then it can check for double-spending. However, if the coin was issued by a remote currency server then the local currency server forwards the coin to the remote one in exchange for a check payable to the local one. This check is then cleared by the global accounting infrastructure.

In step 2 the server returns either newly issued coins or checks that have been cleared by the currency server encrypted with the session key  $SessK_X$ . Encryption with  $SessK_X$  implicitly authenticates CS to  $X$  and provides secrecy between  $X$  and CS.

NetCash allows an exchange between two agents, payor  $A$  and payee  $B$ , where both agents are protected from fraud. In this scheme,  $A$  is guaranteed anonymity together with a valid receipt or its money back, and  $B$  is protected from double-spending.  $A$  obtains a coin triplet  $\langle C_B, C_A, C_X \rangle$  from the currency server CS where each coin is identical in value and serial number. Furthermore each coin is valid for a certain non-overlapping period of time (beginning with  $C_B$  and ending with  $C_X$ ). The first two coins are intended for  $B$  and  $A$  respectively. In the first window of time only  $C_B$  can be used with CS. In the subsequent windows of time,  $C_A$  and  $C_X$  can be used exclusively. Furthermore,  $B$ 's public key is encrypted in coin  $C_B$  (so that only  $B$  can use the coin) and in  $C_A$  (to decrease the amount of state CS has to keep track of to be able to issue  $A$  a receipt). Therefore, if  $A$  wants to pay  $B$  it will ask CS for a coin triplet to be used for  $B$ . Then  $A$  will keep  $C_X$  and  $C_A$  and sends  $C_B$  to  $B$  and waits for  $B$  to send back a receipt. If  $B$  does not give a receipt to  $A$ , "A can query the currency server and check whether  $B$  spent the coin" [12]. If  $B$  spent the coin, CS will give  $A$  a receipt. On the other hand,  $A$  can request a refund during the window when  $C_A$  is valid if the coin was never spent.

## 2.2 NetCheque

NetCheque is a distributed accounting service supporting the credit-debit model of payment [15]. Clients using NetCheque maintain accounts with accounting servers in much the same way that people maintain conventional checking accounts with banks. Similar to a paper check, NetCheque bears an electronic signature, and “must be endorsed by the payee, using another electronic signature, before the check will be paid” [15]. These signatures are authenticated using Kerberos. Furthermore, since NetCheque is a distributed accounting service, various accounting servers arranged hierarchically (see figure 2) settle accounts when endorsed checks are exchanged.

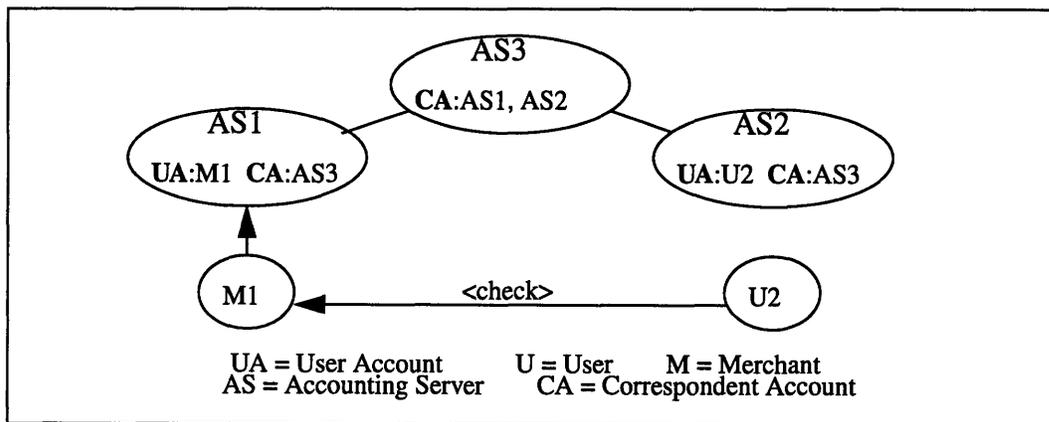


FIGURE 2. Accounting Hierarchy (from [15]).

The check itself contains the currency unit, the amount of the check, an expiration date, an account server identification number, the payee, together with the signatures and endorsements collected during the processing (verifiable by the accounting server against which the check was drawn). Users can write and deposit checks via the **write-check** and **deposit-check** functions respectively and determine their account balances using the **statement** function.

To use **write\_check**, a person passes as parameters the accounts against which the check is to be drawn, the payee, the amount, and the currency unit. This function obtains a Kerberos ticket that will be used to authenticate the payor to the accounting server and produces the cleartext [15]. On the other hand, the **deposit\_check** function reads the cleartext part of the check, obtains a ticket from Kerberos used to authenticate the payee to the payer's accounting server, endorses the check via a special Kerberos ticket called a proxy, and appends the endorsement to the check. Then, an encrypted connection is opened to the payee's accounting server and the endorsed check is deposited. If both the payee and payor have the same accounting server, the accounting server will verify immediately if the check cleared. However, if they do not have the same accounting server the payee's server places a hold on the payee's account until verifying if the check clears. As the check is processed by multiple accounting servers, each server appends its own endorsement to the check. If the check is rejected the payee takes whatever action (s)he deems necessary. However, if the check clears, the payees' account is credited with the appropriate amount.

## 2.3 Millicent

Millicent is a lightweight and secure protocol that enables commercial transactions over the internet [7]. It is designed to support small monetary transactions, including amounts that are smaller than a cent. It is based on the "decentralized validation of electronic cash at the vendor's server without any additional communication, expensive encryption, or off-line processing"[7]. This system makes use of the *scrip* and *broker* abstraction. A *scrip* is vendor-dependent cash and comes in two kinds: vendor scrip and broker scrip. A *broker* is the person who is in charge of account management, billing, and establishing accounts with vendors.

Scrip is similar to cash in the sense that it has intrinsic value. However, it only has value when used with the particular merchant that issued it. In this sense it, is like a calling card or a gift certificate which is merchant-dependent. Like electronic cash, “scrip will consist of a signed message attesting that a particular serial number holds a particular value” [11]. The role of brokers in this system is to serve as accounting intermediaries between vendors and customers. Customers form ties with brokers in the same way that they do with banks and credit card companies. Brokers buy and sell vendor scrip as a service to the vendors and the customers[11]. Furthermore, broker scrip serves as a coherent currency for customers to use when buying scrip from vendors, and for vendors to give as a refund for unspent scrip [11].

In this scheme, authenticity between customer and merchant is guaranteed by a secret key used in an efficient symmetric encryption method such as DES or RC4. Scrip can be used to establish the shared key. The secret key is generated when a customer buys scrip using a secure non-Millicent protocol or purchased using a secure Millicent transaction [11]. Therefore, a user has to keep track of the secret keys it has established with the merchants. For performance a hash, such as MD5, is computed on the plain text of the scrip before signing. It is important that secret keys not be very large because it might decrease performance. However, since the scrip value is so small it is not worth the computational cost to try to break the key.

Two major potential problems are duplicating scrip and scalability of the system. To avoid duplicating scrip, vendors will keep bit vectors corresponding to subranges of the issued serial numbers for scrip until the scrip expires or is spent. Therefore, it is difficult for an adversary to spend the forged scrip since double use of the serial number will be noticed. Another deterrent to scrip forgery lies in the fact that the cost of breaking the protocol is greater than the value of the scrip itself. However, each time a user needs scrip, (s)he has to buy scrip from a broker in real-

time; this implies problems related to scalability. Furthermore, there is no mechanism that allows merchant-independent scrip to be purchased by customers and used in transactions with multiple vendors.

Finally, Millicent does not scale well to large payments since the encryption is not strong enough. In general, it is best suited for transactions of little value. Millicent is also a system that provides anonymity as long as the broker and the vendor do not collude. Overall, this system does a fair job at providing a secure means for transacting small amounts of money but only will exist in small niches of payment strategies because of its restrictions to small transactions.

## **2.4 Visa and Master Card's Secure Electronic Transaction (SET)**

Visa International and Master card announced on February 1, 1996 [25] the development of a single technical standard for safeguarding payment card purchases made over open networks. This standard is being published as an open specification for the industry. The specifications are available to be integrated into any payment service and may be used by software vendors to develop applications. It uses RSA-based public key cryptography to provide confidentiality of information, ensure payment integrity, and authenticate both merchants and cardholders.

SET defines a variety of transaction protocols that use public and private key cryptography to securely conduct electronic commerce. We will describe the following mechanisms:

1. Cardholder registration
2. Merchant registration
3. Purchase request
4. Payment authorization
5. Payment capture.

### 2.4.1 Cardholder Registration

This protocol is defined between the cardholder computer and the certification authority. Cardholders have to register with a Certificate Authority (CA) before they can send requests to merchants. The cardholder initiates the protocol by requesting a copy of the CA's key-exchange certificate. Then the CA responds by transmitting the key-exchange certificate which is used by the cardholder to secure the payment card account number in the registration form request [25]. Then, the cardholder creates a registration form request message, generates a random key to encrypt the registration form request message, and sends both the encrypted request message together with the random key, all encrypted in the CA's public key-exchange key. On receipt of these data, the CA determines the appropriate registration form, digitally signs it by generating a message digest for the form, encrypts it with the CA's private signature key, and sends it to the cardholder. To register an account, the cardholder fills out the registration which contains information such as cardholder's name, expiration date, account billing address, and any additional information the issuing financial institution believes necessary to identify the certificate requester as the authentic cardholder. Then, the cardholder signs the registration information, generates two random symmetric encryption keys, places one key inside the message (one key is used to encrypt the registration and the other will be used by the CA to encrypt the response), and puts the other key together with the account information and a random number (that will be used by the CA to generate the certificate) into a digital envelope.

When the CA receives the cardholder's request, it decrypts the digital envelope to obtain the account information and symmetric encryption key (which it uses to decrypt the registration request). Then, the CA verifies the information contained in the registration request. However, the details of how the CA obtains this information is outside the scope of the protocol. Once the

information has been verified, the CA generates a random number that is used together with the random number produced by the cardholder to generate a secret value. Then, the account information and the secret value are passed through a one-way hash function and the CA signs the cardholder certificate. Next, a response message containing the random number generated and encrypted (with the symmetric key sent by the cardholder in the registration message) by the CA is sent to the cardholder [25].

When the cardholder receives the response from the CA, it stores the certificate for use in future electronic commercial transactions. Next, “the cardholder decrypts the registration response using the symmetric encryption key that it sent to the CA in the registration message” [25]. Finally, it combines the value that was sent in the registration message with the CA’s random number to compute the secret value to use with the certificate.

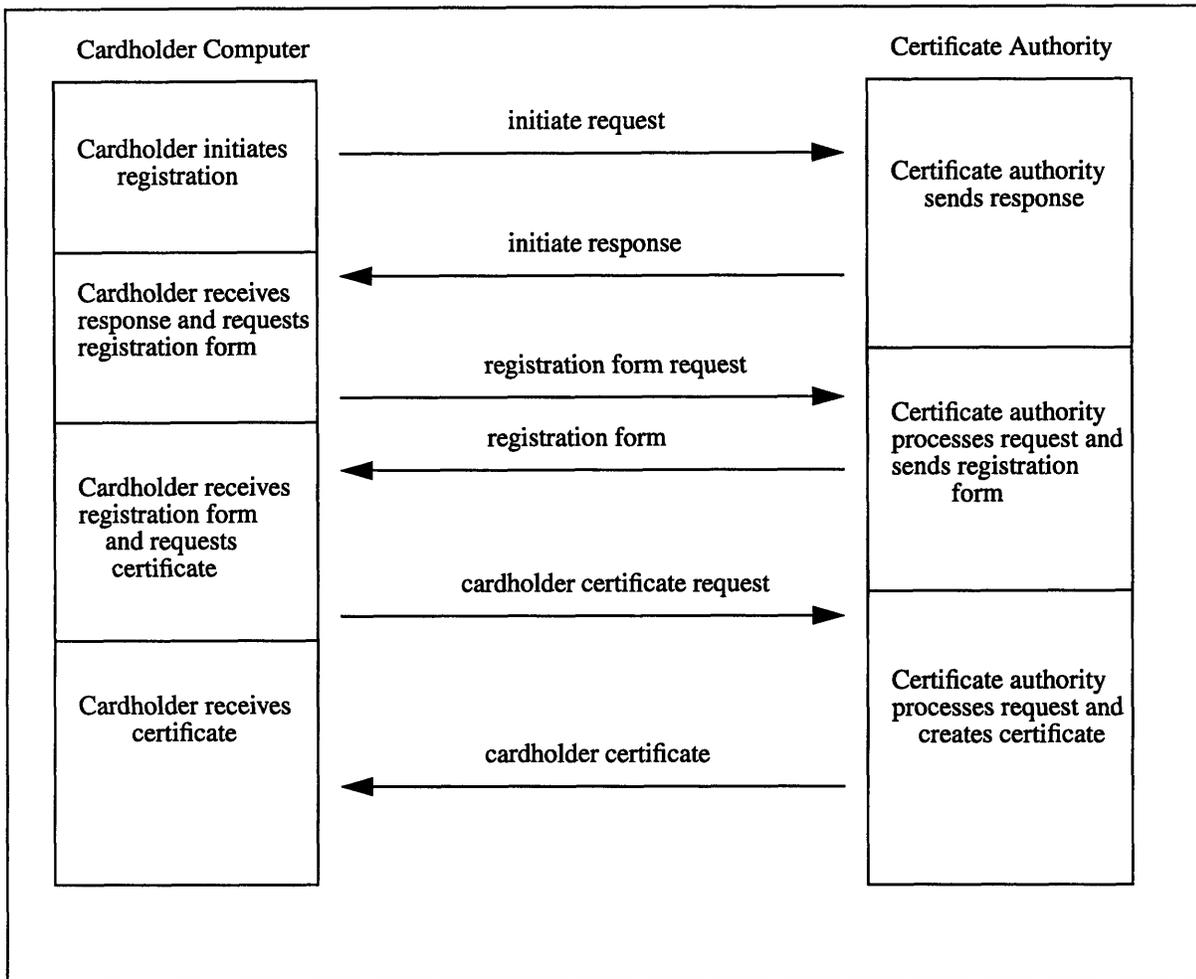


FIGURE 3. Summary of carholder registration protocol (from [25]).

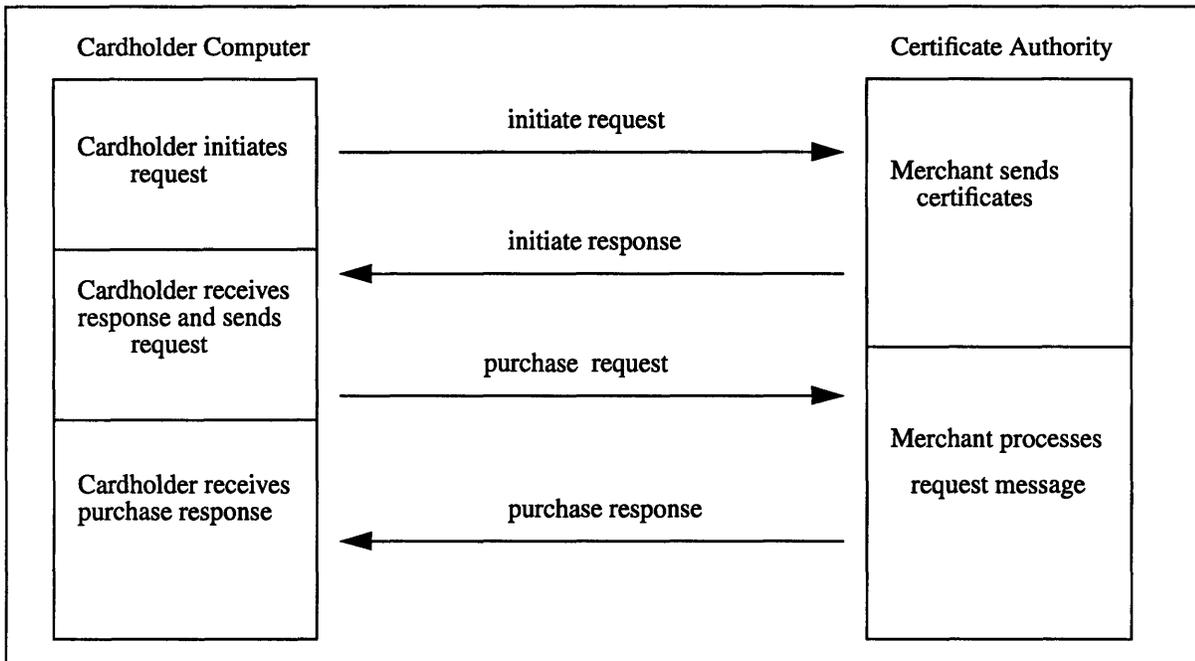
### 2.4.2 Merchant Registration

Merchants register with the CA in a similar fashion to cardholders. The merchant registration mechanism is composed of five parts, instead of the seven that the cardholder registration process involves. The major difference is that after the merchant initiates a request, the CA responds by sending a registration form. Therefore, the merchant does not have to explicitly request a registration form.

### 2.4.3 Purchase Request

This mechanism defines how the cardholder transmits the credit card account information to pay for a given item. It assumes the cardholder and merchant have already registered and that the cardholder has decided which items to purchase. The SET order process is begun when “when the cardholder software requests a copy of the merchant’s and gateway’s certificates” [25]. When the merchant receives the request, it assigns a unique identifier to the message and sends the merchant and gateway certificates.

Upon receipt the cardholder generates the Order Information (OI) and the Payment Instructions (PI) and associates the unique identifier produced by the merchant with the OI and the PI. Next, the cardholder generates a symmetric encryption key used to encrypt the PI and sends the PI together with the OI. Then, the merchant receives the message and processes the order including the payment authorization described in the next section. After the OI has been processed “the merchant software generates and digitally signs a purchase response message, which includes the merchant signature certificate and indicates that the carholder’s order has been received by the merchant” [25]. Then, this response is transmitted to the cardholder.



**FIGURE 4. Summary of purchase request (from [25]).**

#### **2.4.4 Payment Authorization**

This mechanism is defined between the merchant and the payment gateway when the merchant is processing an order from a cardholder. The merchant creates an authorization request which contains information such as the amount to be authorized and the transaction identifier from the OI. Then, the PI and the authorization request are sent to the payment gateway. When the payment gateway receives the PI and the authorization request, the payment gateway verifies their authenticity and “formats and sends an authorization request to the Issuer via a payment system” [25]. Once the payment gateway receives a response from the Issuer it generates an authorization response message which contains an optional capture token (described in the next section). The response is then sent to the merchant. Then, the merchant stores the authorization response and capture token and lastly delivers the goods described in the original form.

## 2.4.5 Payment Capture

This mechanism defines how the merchant receives his payment. There may be a large gap in time between the message requesting authorization and the message requesting payment. Once the merchant initiates this portion of the mechanism, the merchant sends the payment gateway the capture request and optionally the capture token. Furthermore, SET allows for multiple requests to be made in parallel. When the payment gateway receives the capture token (if sent) and the information in the capture request, the payment gateway creates a clearing request which it sends to the Issuer. Then, the payment gateway creates and then signs the capture response message and transmits the message to the merchant. Finally, when the merchant receives the capture response message, it stores the message to be used for reconciliation with payment received from the acquirer.

### **3.0 Motivation**

The previously mentioned systems share something in common: they attempt to describe a framework that supports realtime electronic payments over a network. Furthermore, the commercial transactions are carried out solely in the electronic domain. The fact that these systems are completely paperless is a double-edged sword. The advantage lies in the elimination of the paper processing overhead; however, in the short term, we believe that in daily transactions, such as buying groceries at the supermarket, people will feel more comfortable using paper cash and checks than their electronic counterparts.

Paper-based money has been used for centuries. Since it has been in use much longer, people are much more familiar with this type of currency than any of the electronic versions discussed so far [7][11][12][15]. Therefore, in the short-run, neither customers nor merchants will switch to a completely paperless scheme. In light of this assumption, we propose a system that will allow paper-based commercial transactions, in the form of bank checks, at the front end with autonomous paperless processing thereafter. Thus, our system will serve as a model of transitional technology between the current system, described in the following section, and the previously described systems [7][11][12][15].

#### **3.1 Current Model for Processing Bank Checks**

After a check is deposited at a local bank, it is forwarded to a processing center. At the processing center, the courtesy amount block (where the user has written the quantity of the check in numbers) is read by human operators and the quantity of the check is keyed in by hand into a computer. After this stage, the amount for which the check was written is printed on the bottom of the check in Magnetic Ink Character Recognition (MICR) ink. This ink contains an iron oxide

so that the characters are magnetized and can be “read” by sorting machines and computer input devices. Then, the checks move through a combination of three institutions: correspondent banks, clearinghouses, and Federal Reserve Banks. The correspondent banks handle checking accounts for some banks and provide geographic coverage for others (see figure 5). Clearinghouses “receive the checks banks have credited to their own accounts and sort the checks for distribution to the banks on which they are drawn”[5], while the regional offices of the Federal Reserve Bank serve as clearing houses (see figure 5).

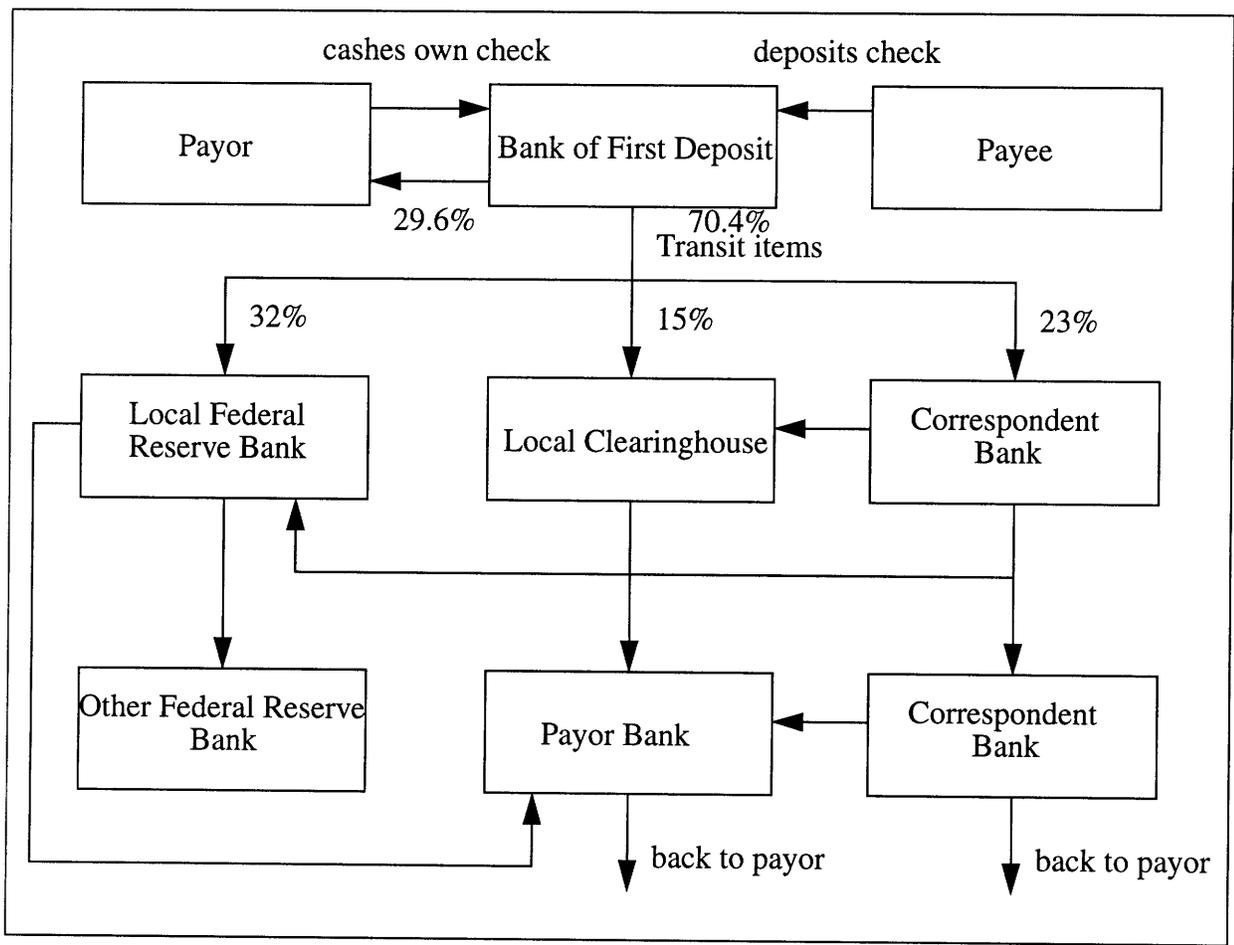


FIGURE 5. Complete check flow (from [5]).

Suppose a consumer pays a merchant with a check from the bank  $B_{user}$  and the merchant deposits the check at the end of the day at bank  $B_{merchant}$ . Since the user's account is not at this location,  $B_{merchant}$  mails the bank check to a processing center where the information on the check is read by hand. Then, the checks are mailed to the federal reserve board  $FRB_{merchant}$  that presides over  $B_{merchant}$ . At  $FRB_{merchant}$  the checks are fed into high-speed MICR readers which read the data, such as user's bank name/ID, the checking account number, and the dollar amount to decide which FRB the check will be mailed to. Once the check reaches  $FRB_{user}$  it is fed into another high-speed MICR reader to determine the bank it needs to be forwarded to. Then, the bank check is sent to  $B_{user}$  and the data detected by the MICR reader is transmitted electronically to  $B_{user}$ .

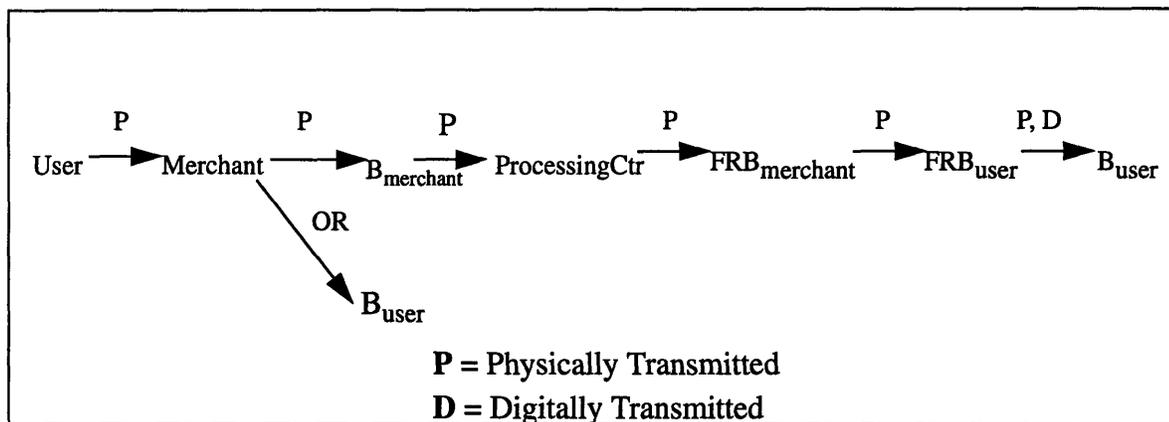


FIGURE 6. Typical check route

### 3.2 Design Goals

We propose a system that decreases the need for human intervention when processing a check. Furthermore, we propose a system that greatly decreases the number of banks the check physically passes through. We will also decrease the amount of processing performed in the ana-

log domain, thus decreasing the total processing latency. Furthermore, we will allow the secure electronic transfer of bank check data, thus eliminating and speeding several of the steps depicted in figures 5 and 6 while maximizing the utilization of the available World Wide Web infrastructure.

## 4.0 *iCheck* Model/Design

Before Tim Berners Lee wrote the first web browser at CERN in 1992, most client/server applications communicated in proprietary formats and non-standard protocols built on top of the Transfer Control Protocol/Internet Protocol (TCP/IP). However, since that time a network service built on top of existing local and wide area networks called the World Wide Web (WWW) has grown to enormous proportions. Clients, called web browsers, on the WWW communicate with web servers via a protocol called Hyper Text Transfer Protocol (HTTP) which is built on top of the reliable transport service provided by TCP [4]. Web browsers, using a Uniform Resource Locator (URL) to identify a resource at the server, request information which is returned in the form of a page of HTML commands. Web browsers parse this stream of commands and use tools from its native operating environment to build display images representing the material in the command stream. The command stream from the server may include text formatting information, embedded graphics and other selectable links. Developers of client/server applications can now take advantage of the available framework the WWW has provided them. For this reason, one design goal of the *iCheck* system is to maximize the utilization of the WWW infra-structure.

Other design goals for our proposed system are to eliminate the need for human intervention for reading the check's courtesy amount and to eliminate the need for the physical transmission of checks between  $B_{\text{merchant}}$  and  $B_{\text{user}}$  in a secure fashion. We call our proposed system *iCheck*. This system is composed of two main modules: the Bank Module (one client per bank) and the Central Storage Facility (one global server).

The Bank Module initiates the processing by scanning the bank check into the electronic domain. This module is responsible for authenticating itself to the Central Storage Facility and securely uploading the scanned check. This is achieved by first negotiating a session key through

a three phase handshake protocol (described in section 4.1.3) and then encrypting the data with the session key. Furthermore, the Bank Module is responsible for securely downloading data from the Central Storage Facility.

The Central Storage facility is responsible for storage, indexing, and retrieval of the bank check data. Furthermore, it is also responsible for maintaining the data safe from potential intruders.

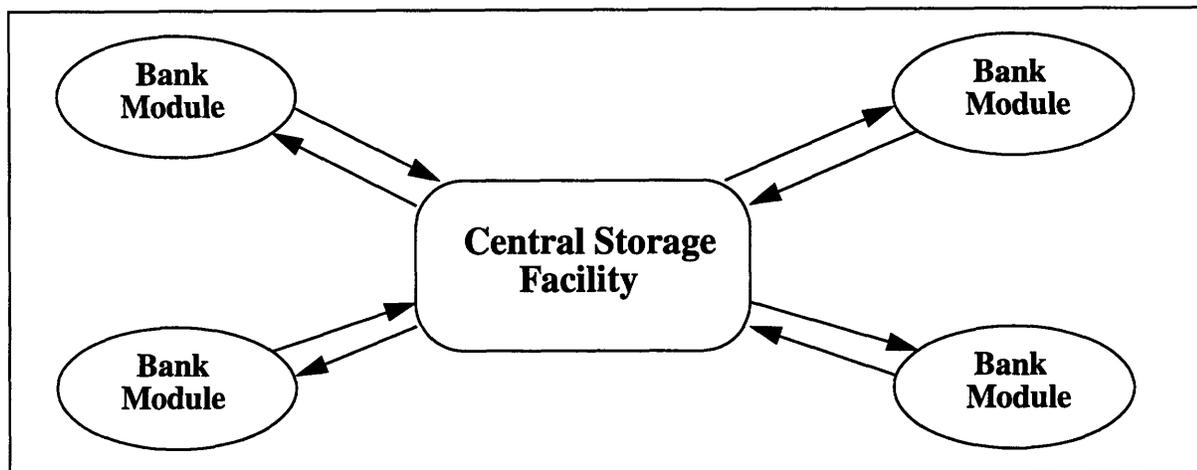


FIGURE 7. *iCheck* Global module view.

## 4.1 Bank Module

The Bank Module is composed of three main parts: Optical Character Recognition (OCR) module, Check and Briefcase Object Packer module, and the security module. *iCheck* processes bank checks through a set of three steps as follows:

1. Optical character recognition of the check's courtesy amount.
2. Creation of Briefcase Objects and Check Objects.
3. Establishment of a secure connection with a remote bank and transmission of the check image together with the recognized data.

### 4.1.1 OCR Module

This module is responsible for the automated reading of handwritten/typewritten amount on the courtesy block using neural network techniques after the check has been scanned in. It is divided into three main stages: preprocessing, recognition, and postprocessing [8, 26]. The preprocessing stage itself consists of: a segmentation of the characters into individual units, normalization, slant correction, thinning, and thickening. The recognition stage consists of a neural network which takes the preprocessed data as an input to recognize the characters. Finally, postprocessing involves the examination of the numeral bitmap and output of the neural net to produce confidence measures about the predicted value.

The first step in preprocessing, called segmentation, separates alphanumeric symbols into individual units. Based on general location and size within segment image, commas and other punctuation marks are identified. Then, each digit is normalized to a 16 by 16 bit grid followed by a de-slanting procedure which decreases the angle of slant [13, 14]. Thinning turns the numeral into a one pixel thickness skeleton [9, 14]. A good skeleton “retains the connectivity and structural features of the original pattern and, once defined, is then thickened to a thickness of two pixels” [26].

In the recognition phase the 16 by 16 bit image is passed into a neural network based recognizer. The neural net, consisting of 256 input nodes and 40 hidden nodes is trained on extensive amounts of data available from the National Institute of Standards and Technology (NIST) database. The output consists of 10 nodes which are mapped to the 10 different digits.

In the postprocessing stage “each segment is passed through a second neural network that has been trained with negative templates” [26]. Following the recognition stage, the output from

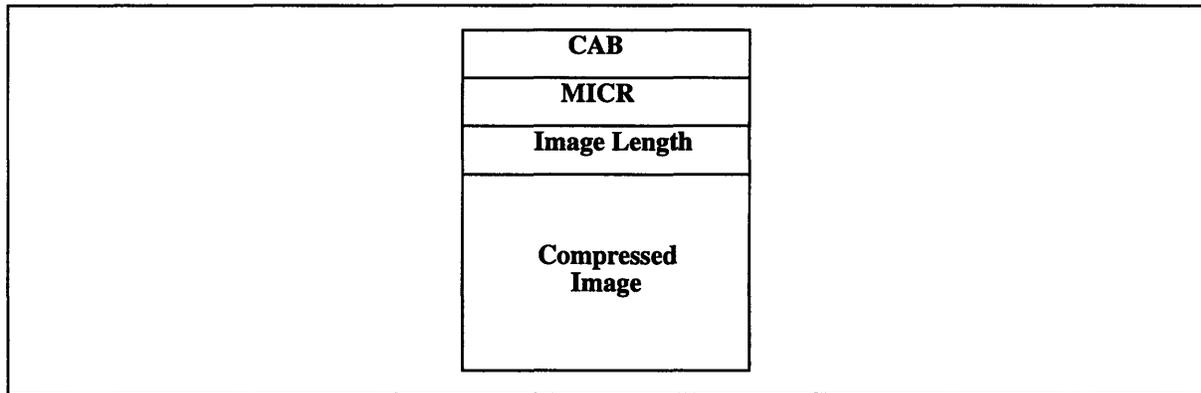
the two networks are compared. If the outputs are not consistent the data is input back into the segmentation for a second pass.

---

#### **4.1.2 Check and Briefcase Object Packer Module**

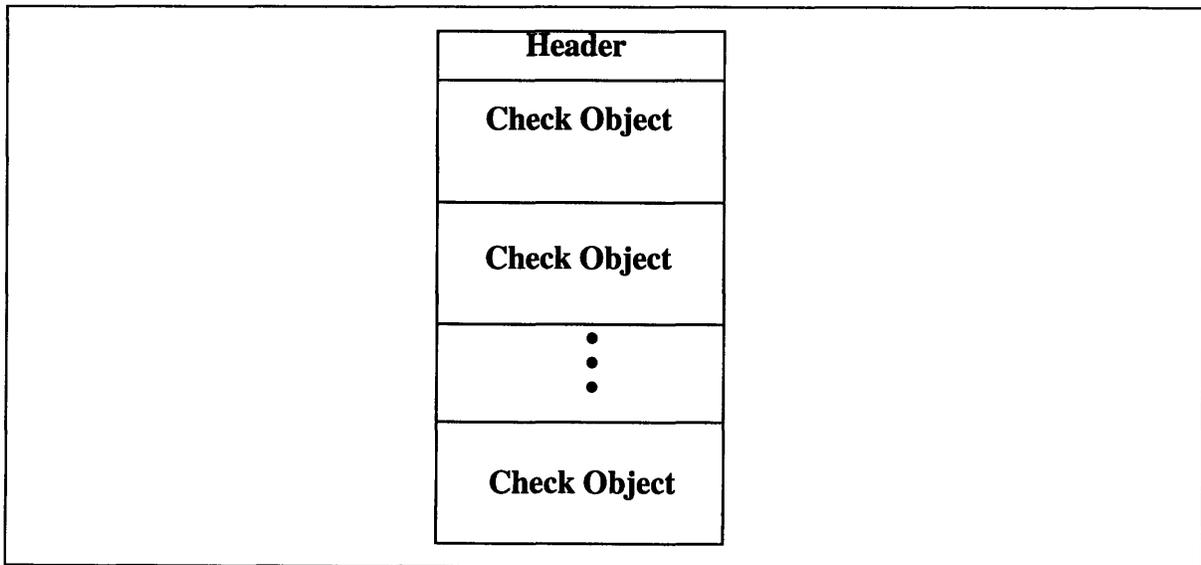
For several years industry and academia have been moving from procedural centered programming to object oriented programming [22]. One idea from object oriented programming that we decide to use in the Object Packer Module is data encapsulation since it is more logical and neat to group together data that is semantically related. This way, information such as the CAB and MICR are bound together in one entity: the Check Object inside a Briefcase Object.

This module is responsible for building the check and briefcase objects after the OCR module recognizes and stores the Courtesy Amount Block (CAB) and MICR quantities as temporary variables. Then, the raw check image is JPG compressed and the new size of the image is also stored which simplifies the Central Storage Facility parser (see section 4.2.1). It is important to note that the image must be stored at a high enough resolution so that it is legally acceptable (i.e., we will be able to recognize the signature). However, excessive detail stored in the image will degrade the network transmission performance due to the increased bandwidth required to transmit more data. Therefore, a careful trade-off needs to be made between image resolution and transmission speed. Finally, the information in the temporary variables is used to create a Check Object by storing the CAB, MICR, image length, and JPG image together.



**FIGURE 8. Check Object.**

Each time a check is processed, a new Check Object is created and appended onto an entity we call “Briefcase Object”. The Briefcase Object is composed of a header followed by the Check Objects. The header contains information such as the processing bank name and the number of Check Objects in the briefcase. We send a Briefcase Object, and not individual Check Objects, for efficiency. In this way, the cost of transmission is amortized over the number of checks in the briefcase.



**FIGURE 9. Briefcase Object.**

A limit of 4 gigabytes was established for the length of the Briefcase Object due to the limitations of form-based file uploading (the mechanism by which we upload the Briefcase to the Central Storage Facility)[6, 23]. To enforce this size limitation, we check for the size of the current Briefcase. If the difference between 4 gigabytes and the current size is at least the size of the current Check Object, the Check Object is appended to the Briefcase. Otherwise, a new Briefcase is created.

### **4.1.3 Network/Security Module**

The goal of this module is to establish a secure connection between a Bank Module and the Central Storage Facility to transmit the Briefcase Object. A secure connection is defined as a connection where the data are privately transmitted and both the sender and the receiver are mutually authenticated to each other. In other words, an eavesdropper that listens to the communication channel between the Bank Module and the Central Storage Facility will not be able to decipher the contents of the transmitted data. Furthermore, the Bank Module will be sure that it is communicating with the Central Storage Facility and vice-versa.

Various factors affect our design decisions: encryption algorithm computation time, algorithm robustness, and available cryptographical application programming interfaces (API). Only by carefully examining these factors are we able to achieve our goal of secure transactions.

A factor that influences our design criteria is the time our encryption algorithms takes to execute. We decide to use a public key algorithm to encrypt a session key which is later used to encrypt the data using a private key algorithm (see section 4.1.3.1). The reasoning behind this decision is based on the fact that in general, public key algorithms are approximately two to three orders of magnitude slower than their private key counterparts[24] and the session key is orders of

magnitude smaller than the data to be encrypted. For example on a '90 MHz Pentium, RSA Data Security's cryptographic toolkit BSAFE 3.0 has a throughput for private-key operations of 21.6 Kbits per second with a 512-bit modulus and 7.4 Kbits per second with a 1024-bit modulus"[24]. By comparison, a private key algorithm such as DES is generally at least 100 times as fast as RSA when implemented in software[24]. Furthermore, the encryption algorithm computation time is proportional not only to the key modulus, but to the length of the input data. Therefore, the decision to use private key algorithms to encrypt the data is further substantiated since the length of the data encrypted in *iCheck* can be up to 4 gigabytes long (orders of magnitude longer than the key modulus).

We consider a number of public and private key algorithms that have been shown to be secure by industry standards [10, 20]. The two public key algorithms we consider are El-Gamal [20] and RSA. RSA's security is contingent on the difficulty of factoring large composite numbers while El-Gamal's is contingent on the difficulty of solving the discrete logarithm problem [20]. The three private key algorithms we consider are DES, RC2, and RC4.

DES is the Data Encryption Standard, an encryption block cipher defined and endorsed by the U.S. government in 1977 as an official standard; the details can be found in the latest official FIPS (Federal Information Processing Standards) publication concerning DES [21]. It specifies a cryptographic algorithm to encrypt and decrypt 64-bit blocks of data under the control of a unique key as defined in Federal Information Processing Standards (FIPS).

RC2 is a variable key-size block cipher. It is faster than DES and is designed as a replacement for DES. It can be made more secure or less secure than DES against exhaustive key search

by using appropriate key sizes [24]. It has a block size of 64 bits and is about two to three times faster than DES in software. The algorithm is confidential and proprietary to RSA Data Security.

RC4 is a variable key-size stream cipher with a byte-oriented operations. The algorithm is based on the use of a random permutation. Furthermore, eight to sixteen machine operations are required for every output byte, and the cipher can run very quickly [19]. Although the algorithm is proprietary, it has been scrutinized under conditions of non-disclosure by independent analysts and it is considered secure [24].

One practical factor that affects the decision on how to implement our Network/Security Module is the availability of cryptographical application programming interfaces for the previously mentioned algorithms. We decided to use CryptoAPI: a cryptographical API by Microsoft for various reasons. CryptoAPI provides system level access to common cryptographical functions such as key generation, key exchange, data encryption and decryption, hashing and digital signatures. Also, CryptoAPI eases the process of making applications and cryptographic services modular. This allows software developers that use CryptoAPI in their applications and to substitute-in stronger Cryptographical Service Providers (CSP) modules as the need may arise. The CryptoAPI programming model is similar to the Windows Graphical Device Interface (GDI) model in that the CSP's are to CryptoAPI as device drivers are to the Windows GDI. Therefore, just as "well-behaved" applications are not allowed to communicate directly with the hardware, "well-behaved" applications cannot directly access the CSP's.

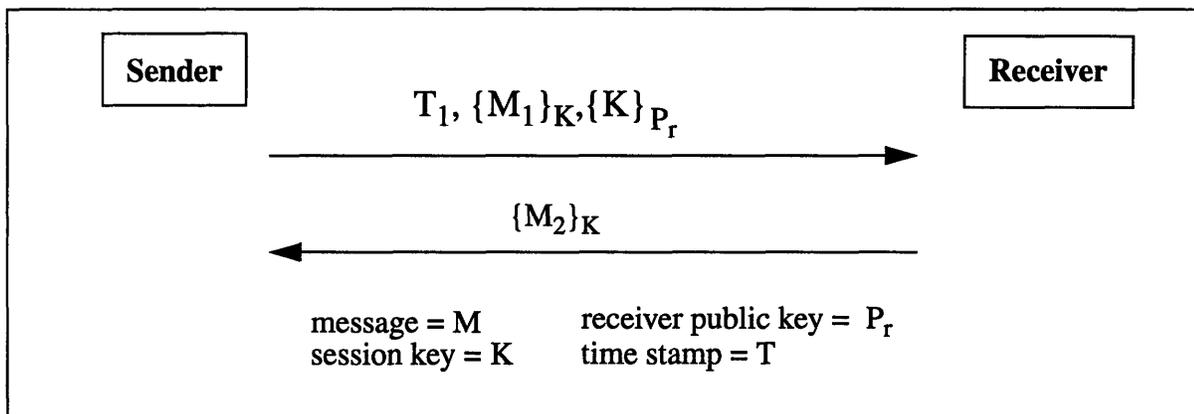
After taking into consideration both public key algorithms and the choice of CryptoAPI, we determined that the best design decision is that RSA be used as the public key algorithm for *iCheck* [6, 10]. Furthermore, after considering the various private key algorithms, we decided to

use RC4 since it is both robust and quick by comparison to DES and RC2 (on a 120 MHz Pentium based computer DES encrypts 1,138,519 bytes a second, RC2 encrypts at 286,888 bytes a second, while RC4 encrypts at 2,377,723 bytes a second) [19].

#### **4.1.3.1 Session Key Exchange Protocol.**

As mentioned earlier, we decided to use a public key algorithm to encrypt a session key which is later used to encrypt the data using a private key algorithm since public key algorithms are in general much slower than their private key counterparts [24]. We explored two possible ways to securely exchange a session key: the nonce session key exchange protocol and the three-phase session key exchange protocol [19].

We first explored the nonce session key exchange protocol. In this protocol, there is a sender that wants to give a message to a receiver. The sender creates a random session key and encrypts the message with it. Then the sender encrypts the session key with the receiver's public key and sends it to the receiver together with the encrypted message. When the receiver gets the data it can decrypt the session key with its private key and then use the session key to decrypt the message. This protocol is vulnerable to one form of attack: replay of transmitted data. In other words, an eavesdropper that acquires copies of one or more encrypted messages and the encrypted keys could at some later time send one of these messages to the receiver and the receiver will have no way of knowing the message did not come directly from the original sender. To reduce the risk of attack, we timestamp all the messages and have the receiver verify that the messages are current. In case of a replay attack the receiver would notice this since the timestamp would have expired.



**FIGURE 10. Nonce session key exchange protocol.**

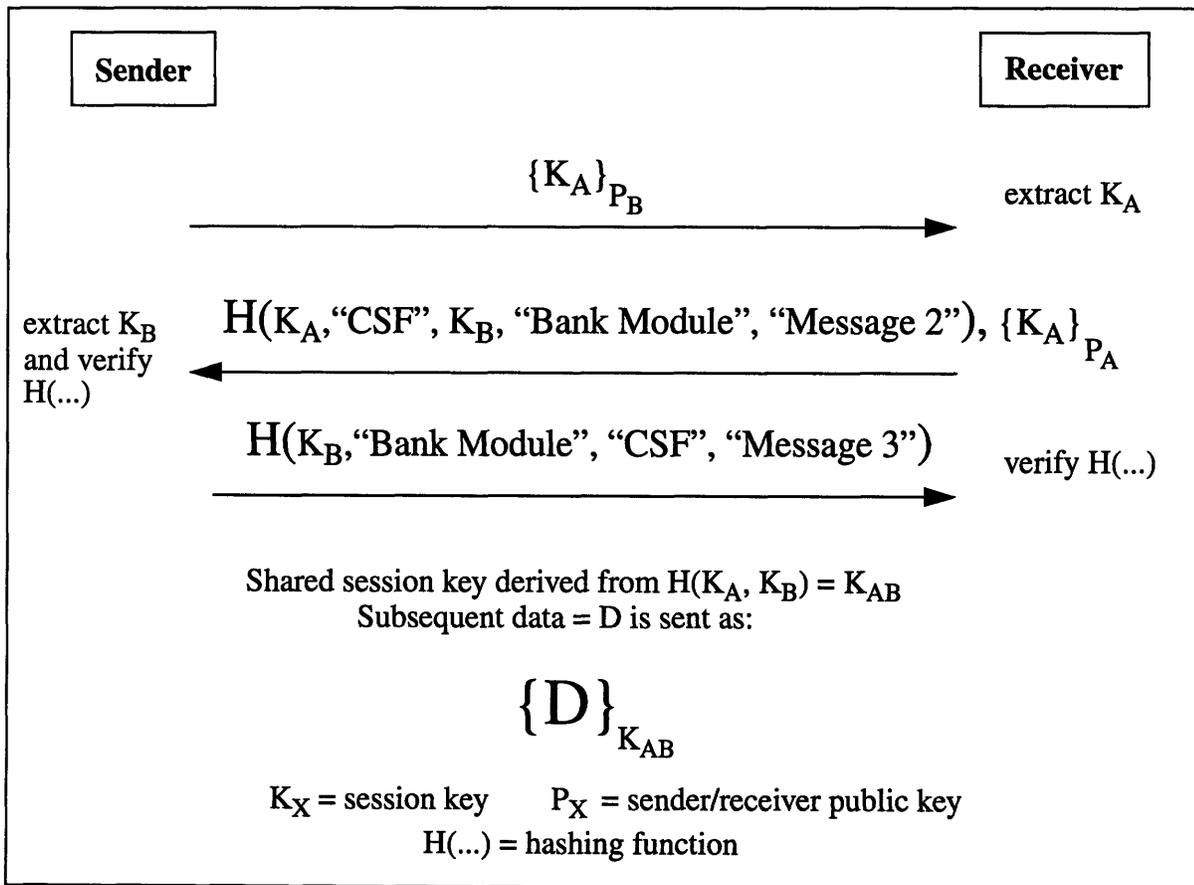
However, we decided to use the three-phase session key exchange protocol since it provides a way for two parties to create an authenticated, real-time connection between themselves without the need for time stamps [19]. The end result of this protocol is a session key that is shared by both of the parties involved. This protocol is known as a three-phase protocol because it requires that the two parties exchange three packets of data in the process of creating the shared session key. This protocol eliminates, instead of reducing, the risk of replay attack. Here, both parties contribute a portion of the data that the negotiated session key is derived from. This protocol ensures that both the parties are current and are sending messages directly to each other. We make the assumption that both parties involved already possess their own set of public/private key pairs and that they have also obtained each other's public keys. Furthermore, we assume that the parties have already exchanged human-readable user names. This is usually done at the same time the public keys are exchanged, since the user name is included as part of each certificate (see section 4.1.3.2).

In the first phase of the protocol, the sender creates a random session key  $K_A$ . The sender then encrypts  $K_A$  with the receiver's exchange public key  $P_B$ . The encrypted  $K_A$  is then sent to the receiver. Upon receipt, the receiver decrypts  $K_A$  with the receiver's exchange private key  $S_B$ .

In the second phase of the protocol, the receiver creates a random session key  $K_B$  of its own. The receiver then encrypts  $K_B$  with the sender's exchange public key  $P_A$ . The encrypted  $K_B$  is then sent to the sender. The receiver then computes the hash  $H_1$  of  $K_A$ , the receiver's name,  $K_B$ , the sender's name, and the text "Message 2" via the MD5 hashing algorithm [24]. This hash value is then sent to the sender. The data must be hashed in the standard sequence, so the sender will be able to properly validate it. Then the sender decrypts  $K_B$  with the sender's exchange private key  $S_A$ . The hash value  $H_1$  is also received. The sending user then validates the receiver's hash value  $H_1$  by creating a hash of its own containing the same data, and comparing the two hash values. If the hash values do not match, then either the destination user has not been forthright, or someone else is tampering with the data between the two parties. In either case, the protocol is terminated and the communication link severed. If the two hash values do match, this tells the sender that the destination user is presently on-line and in real-time communication. This is primarily because the hash value contains  $K_A$ , which was sent out encrypted with the receiver's public key  $P_B$ . Only the real destination user could have decrypted the session key and built the hash value. Having the human-readable user names in the hash makes it possible to involve the users in the process as an additional check [19].

In the third phase of the protocol, the sender builds up a hash value  $H_2$  containing  $K_B$ , the sender's name, the receiver's name, and the text "Message 3." This hash value is then sent to the destination user. The destination user accepts the hash value from the sender and validates it by

creating a hash of its own and comparing the two hash values. If the hash values do not match, then the protocol should be terminated and the communication link severed. If the two hash value do match, this tells the receiver that the sender is presently on-line and in real-time communication. This is primarily because the hash value contains session  $K_B$ , which was sent out encrypted with the sending user's public key  $P_A$ . Only the real sending user could have decrypted the session key and built the hash value.



**FIGURE 11. Three phase session key exchange protocol.**

Once the two parties have exchanged session keys and hash values and the hash values have been properly validated, the protocol is complete. Each party can now independently create a shared session key based on  $K_A$  and  $K_B$  that can be used to send encrypted messages to each

other. To create the shared session key, each party must create a hash session key  $K_A$  and session key  $K_B$  and then derive the key based on the value of the hash. The implementation details of this procedure are outlined in the following paragraphs.

The three phase session key exchange protocol is implemented as a web browser helper application that is launched before data is uploaded from the bank running the *iCheck* Bank Module. First, the helper application calls **CryptGenKey** to generate the random session key  $K_A$  and calls **CryptExportKey** to encrypt the key  $K_A$  with the Central Storage Facility's (CSF) public key  $K_B$ . Then, the helper application opens a socket connection to CSF, sends the encrypted key  $K_A$  to it, and listens to the specified socket for a response from the CSF. The helper application verifies the validity of the hash value  $H_1$  upon receipt of the response and extracts, using the **CryptImportKey** function, the random session key  $K_B$  produced by the CSF. If the hash value is invalid the protocol is terminated and the communication link severed. Finally, the helper application builds up a hash value  $H_2$  and sends it to the CSF through the already opened socket connection. Below is pseudo code that describes the first portion of the protocol up to the point where the helper application verifies the validity of hash value  $H_1$  sent by the CSP (for all the implementation details of the protocol, please see Appendix A).

```
// Get handle to the Cryptographical Service Provider.
CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_FULL, 0);
// Obtain the destination user's exchange public key. Import it into
// the CSP and place a handle to it in 'hDestPubKey'.
CryptGetUserKey(hProv, AT_KEYEXCHANGE, &hDestPubKey);

// Create a random session key  $K_A$ .
CryptGenKey(hProv, CALG_RC4, CRYPT_EXPORTABLE, &hKeyA);

// Export session key  $K_A$  into a simple key blob.
dwBlobLen = BLOB_SIZE;
```

```

CryptExportKey(hKeyA, hDestPubKey, SIMPLEBLOB, 0, pbKeyBlob, &dwBlobLen);

// Send key blob containing session key  $K_A$  to the CSF.
sendto (connectedport, pbKeyBlob, dwBlobLen, 0,
        (struct sockaddr *) &clientaddr, sizeof (clientaddr))
// Receive a key blob containing session key  $K_B$  from the CSF
// and place it in 'pbKeyBlob'. Set 'dwBlobLen' to the number
// of bytes in the key blob.
lettersent= recv (connectedport, pbKeyBlob, MAXLEN, 0)
dwBlobLen=lettersent;

// Receive a hash value  $H_1$  from the CSF and place it in
// 'pbHashValue'. Set 'dwHashLen' to the number of bytes in the hash
// value.
lettersent= recv (connectedport, pbDestHash, MAXLEN, 0)
dwDestHashLen=lettersent;
// Import the key blob into the CSP.
CryptImportKey(hProv, pbKeyBlob, dwBlobLen, 0, 0, &hKeyB)

// Verify hash value received from the destination user.

// Create hash object.
CryptCreateHash(hProv, CALG_MD5, 0, 0, &hHash);
// Add session key A to hash.
CryptHashSessionKey(hHash, hKeyA, 0);
// Add destination user's name to hash.
CryptHashData(hHash, pbDestName, dwDestNameLen, 0);
// Add session key B to hash.
CryptHashSessionKey(hHash, hKeyB, 0);
// Add sending user's name to hash.
CryptHashData(hHash, pbSendName, dwSendNameLen, 0);
// Add "Message 2" text to hash.
CryptHashData(hHash, "Message 2", 9, 0);
// Complete the hash computation and retrieve the hash value.
dwHashLen = HASH_SIZE;
CryptGetHashParam(hHash, HP_HASHVALUE, pbHash, &dwHashLen, 0);
// Destroy the hash object.
CryptDestroyHash(hHash);

// Compare the hash value received from the destination user with
// the hash value that we just computed. If they do not match, then
// terminate the protocol.

if(dwHashLen!=dwDestHashLen || memcmp(pbHash, pbDestHash, dwHashLen)) {
    printf("Key exchange protocol failed in Second Phase!\n");
    printf("Aborting protocol!\n");
    return;
}

```

### 4.1.3.2 Certification Authority.

Certificates are digital documents attesting to the binding of a public key to an individual or other entity. They allow verification of the claim that a given public key does in fact belong to a given individual. Certificates help prevent someone from using a phony key to impersonate someone else. In *iCheck* public key certificates are composed of a public key, name of the owner of the public key, and a hash of the public key and name of the owner. All of this is encrypted with the public key of an entity called a certifying authority (CA).

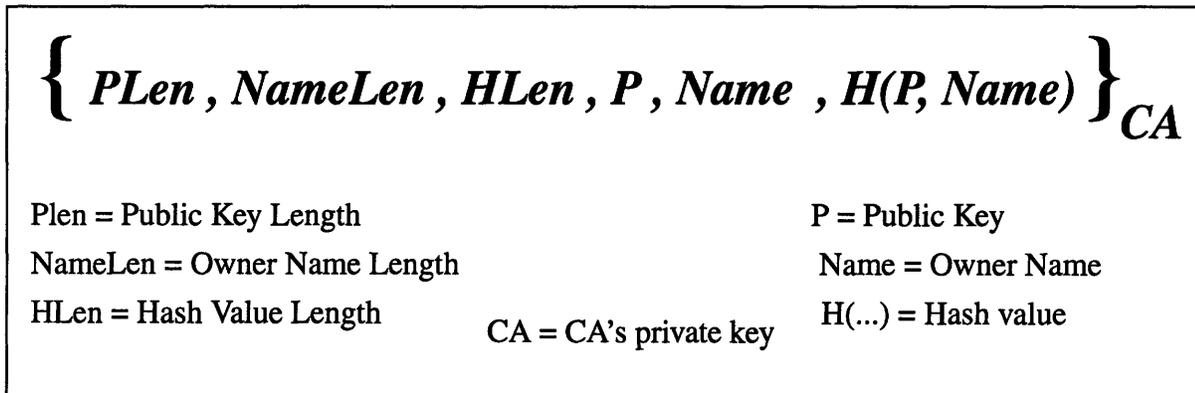


FIGURE 12. *iCheck* public key certificate.

In *iCheck* a trusted third party vouches for the authenticity of the public key keys used to negotiate the session key in the three phase session key exchange protocol. The CA provides an electronic certificate that vouches for the fact that a public key is owned by the sender and the receiver of the three phase session key exchange protocol. This electronic certificate, itself digitally signed by the CA, is stored by the sender and the receiver. Before initiating the three phase session key exchange protocol both the sender and the receiver exchange their public key certificates. Upon receipt, the receiver uses the certificate to verify the sender's public key. At that point the receiver is sure of three things:

1. the original data was not altered (data integrity)
2. the message could only have been encrypted by the holder of that public key (entity authentication)
3. a trusted third party has vouched for the fact that the sender is in fact the holder of that key

Therefore, the strength of the public key certificate provides an acceptable level of assurance to the sender that it holds a copy of the receiver's public key and vice-versa.

It is important to note that there may be a need for multiple CA's. Suppose that the sender gives a certificate to the receiver. The receiver would verify the certificate using the certifying authority's public key and, now confident of the public key of the sender, verify the message's signature. There may be two or more certificates enclosed with the message, forming a, hierarchical chain, wherein one certificate testifies to the authenticity of the previous certificate. At the end of a certificate hierarchy is a top-level certifying authority, which is trusted without a certificate from any other certifying authority. The public key of the top-level certifying authority must be independently known by being widely published. It is also important to note that the more familiar the sender is to the receiver of the message, the less need there is to enclose, and to verify, certificates. For example, if the Bank Module sends messages to the CSF every day, the Bank Module can enclose a certificate chain, which the CSF verifies on the first day. The CSF thereafter stores the Bank Module's public key and no more certificates or certificate verifications are necessary. A sender whose company is known to the receiver may need to enclose only one certificate (issued by the company), whereas a sender whose company is unknown to the receiver may need to enclose two certificates. In general it is good to enclose just enough of a certificate chain so that the issuer of the highest level certificate in the chain is well-known to the receiver. If there are

multiple recipients, then enough certificates should be included to cover what each recipient might need.

## 4.2 Central Storage Facility Module

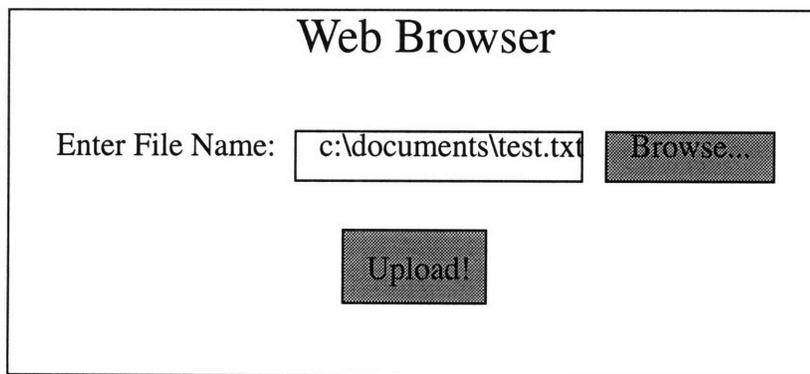
The size of magnetic storage has been steadily increasing and will continue to increase at an exponential rate according to Moore's Law. Therefore, prices have decreased at similar rate for a given amount of magnetic storage. Magnetic storage decreases in cost primarily have occurred because of improvements in three major areas [18]:

1. Magnetic disk storage cost/bit and absolute size
2. Network bandwidth increases which now permits moving data from the storage facility to the client within a human reaction time
3. Display technology

It is now feasible to be able to store terabytes of data in one location. Examples of such systems include the UC Berkeley's digital library research project having a capacity of more than six terabytes, large collections of survey data for use by economists and social scientists covering a wide range of sources such as government censuses and consumer surveys [2], and the Massive Data Analysis Systems at the San Diego Supercomputer Center [3]. This is why we make the design decision that all the check data processed in the OCR Module be stored in one location: the Central Storage Facility (CSF). Therefore, we need a mechanism for uploading data onto the CSF. The mechanism we use is called form-based file uploading and is available on web clients.

Currently, HTML forms allow the producer of the form to request information from the user reading the form. These forms have proven useful in a wide variety of applications in which input is necessary. However, this capability was limited because HTML forms did not provide a

way to ask the user to submit files of data until recently. Service providers who needed to get files from the user had to implement custom user applications. However, the mechanism necessary for client side file uploads is now possible with the advent of form-based file uploading [23]. To have form based file uploading available on a certain HTML file the software developer edits the HTML document to include the form tag **enctype = "multipart/form-data"**, **method = "post"**, and **input type = "file"**. Anyone who views the given HTML file with a web browser will see something similar to the figure below. A user simply needs to enter the location of the file (s)he wants to upload and then selects the Upload button.



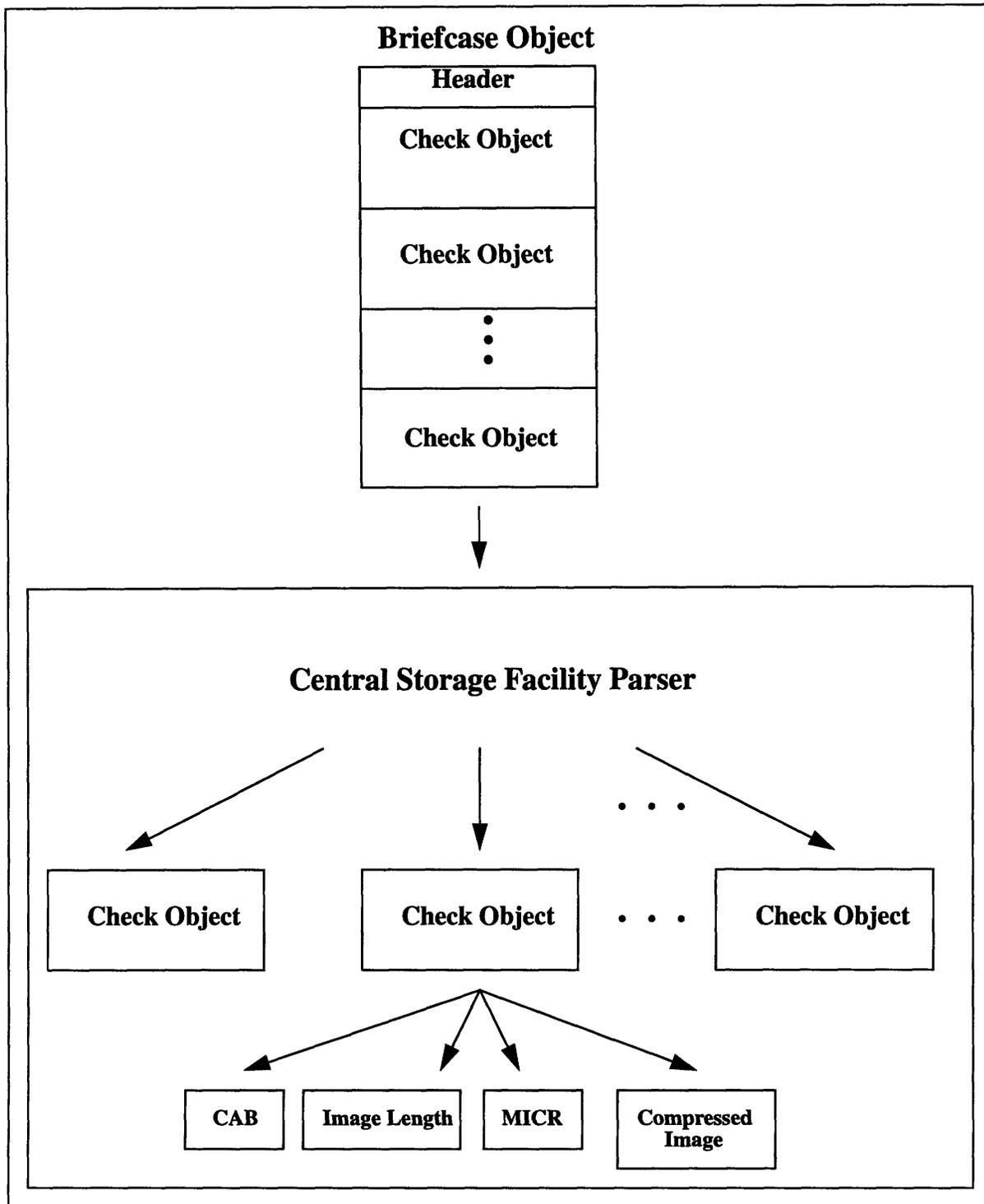
The image shows a window titled "Web Browser". Inside the window, there is a form with the following elements: the text "Enter File Name:" followed by a text input field containing "c:\documents\test.txt", a "Browse..." button, and an "Upload!" button.

**FIGURE 13. Form-based file uploading.**

#### **4.2.1 Parser**

The Parser is the part of the CSF Module that is responsible for processing data (in this case the Briefcase Objects) that have been uploaded from with the form-based file uploading mechanism. The first thing the parser does is to read the **Header** field of the Briefcase Object to extract the identification of the bank that is uploading the data and to extract the field that indicates how many Check Objects are in the Briefcase Object. Then, for every Check Object the

Parser extracts the **CAB**, **Image Length**, **MICR**, and the **Image** and creates a HTML document that contains all of the extracted data and the image.



**FIGURE 14. Central Storage Facility parser.**

Following is a code segment for the CSF parser:

```
fpUploaded = fopen(value[1], "rb");
fscanf(fpUploaded,"%s", strBriefcaseID); //get ID for briefcase
while(fscanf(fpUploaded,"%s", strCAB) != EOF) {
    sprintf(strImagefname, "image%d.pgm", i);
    sprintf(strImagefnamepath,
"c:\\inetpub\\wwwroot\\netbank\\conversions\\%s",strImagefname);

    fpimage = fopen(strImagefnamepath, "w");
    _setmode( _fileno( fpUploaded ), _O_BINARY );
    fscanf(fpUploaded, "%c", &byte);
    for(ibytecount=0; ibytecount<GREYMAPSIZE; ibytecount++) {
        fscanf(fpUploaded, "%c", &byte);
        fprintf(fpimage, "%c", byte);
    }
    sprintf(strBatCommand, "convert %s", strImagefname);
    system(strBatCommand);
    fclose(fpimage);
    //_unlink(strImagefnamepath);

    //Create HTML file
    sprintf(strHtmlfname,
"c:\\inetpub\\wwwroot\\netbank\\conversions\\converteddata%d.html",i);

    fpHtmlfname = fopen(strHtmlfname,"w");
    fprintf(fpHtmlfname, "%s", strheadhtml);
    fprintf(fpHtmlfname,
"<B>BriefCaseID = %s</B><P>\n\n", strBriefcaseID);

    fprintf(fpHtmlfname, "<B>NumChk = %d</B><P>\n\n", i);
    fprintf(fpHtmlfname, "<B>CAB = %s</B><P>\n\n", strCAB);
    fprintf(fpHtmlfname,
"<HR><IMG SRC='\"check%d.jpg'\"><P><HR></BODY></HTML>",i);

    fclose(fpHtmlfname);
    i++;
    _setmode( _fileno( fpUploaded ), _O_TEXT );
}
fclose(fpUploaded);
```

## 5.0 iCheck Scenario

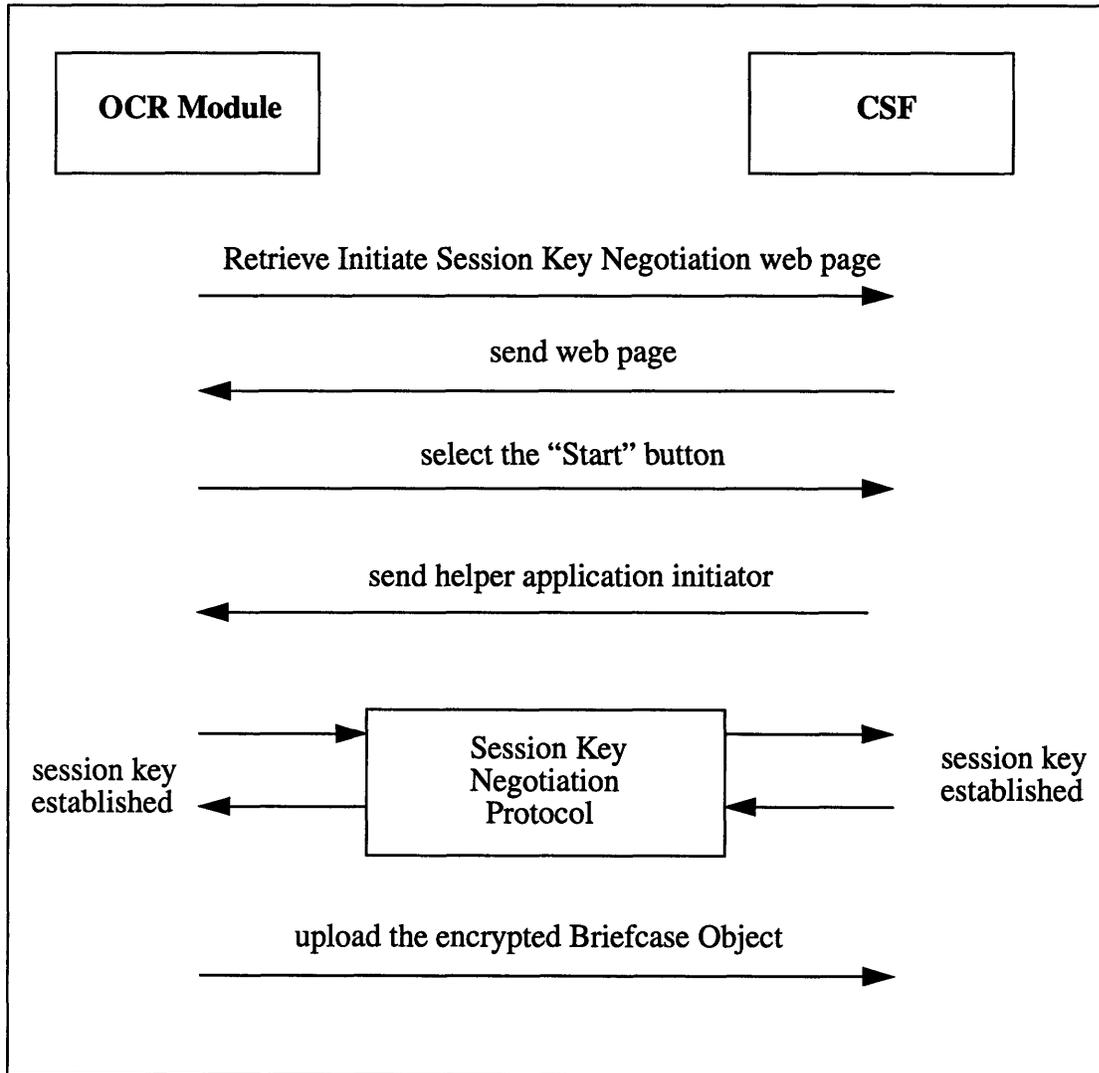
Suppose you are at a grocery store and you write a check to pay for the food you just bought. At the end of the day the merchant deposits the checks that (s)he has gathered for the day into his/her bank account at  $B_{\text{merchant}}$ . The merchant's bank collects the checks and feeds them into the *iCheck* scanner which converts them into digital images. Then the OCR module recognizes the quantity each check was written for and stores the data together with the digital image into a Check Object and appends it onto a Briefcase Object. At certain periods of the day an authorized bank employee connects to a "Initiate Session Key Negotiation" web page provided by the Central Storage Facility (CSF). The bank employee selects the "Initiate Session Key Negotiation" button and waits for a confirmation message. Here, a session key is negotiated between the CSF and the web browser helper application resident in the *iCheck* Bank Module. Then, an encrypted connection between the Bank Module and the Central Storage Facility is opened and the data are transmitted. Thereafter, the uploaded data is available for other banks to download.

The image shows a web browser window titled "Web Browser". Inside the window, there are three distinct sections. The top section contains the text "Initiate Session Key Negotiation" followed by a shaded button labeled "Start!". The middle section contains the text "Enter File Name:" followed by a rectangular text input field and a shaded button labeled "Browse...". The bottom section contains a shaded button labeled "Upload!".

FIGURE 15. Initiate Session Key Negotiation web page.

When a user wants to download check data from the CSF (s)he goes to the “Download Check Data Web” page. There (s)he finds another initiate session key button (used to encrypt the data to be downloaded) and a pointer to the Secured Directory which will contain the encrypted check data.

If a user contests the amount written on the check, the user can request a copy from the bank. The bank will retrieve the digitized image and print a hard copy for the user. Since the digital image was stored at a high enough resolution for legal acceptability, the hardcopy is as good as the original.



**FIGURE 16.** Summary of the *iCheck* system communication between the OCR module and the Central Storage Facility.

## 6.0 Conclusion

### 6.1 How *iCheck* Succeeds to Meet Design Expectations

*iCheck* is a system architecture that securely allows the banking system to increase the check processing speed and decrease the check processing cost. The *iCheck* architecture provides a secure, reliable, and widely available infrastructure for accessing the existing bank payment system over the internet. This is accomplished by designing, developing and integrating the components necessary to decrease bank check processing time and to reduce the need for human intervention in a secure form over open public networks. These components are implemented in two main modules and demonstrate the interoperability and the infrastructure that can benefit banks by allowing them a consistent, secure, trusted way of offering faster and cheaper services. This infrastructure should also enable the offering of innovative new systems which take advantage of new developments in telecommunications technology and enhancements to the existing banking payment system.

#### 6.1.1 Drastic Increase in Speed of Check Processing.

In current banking systems a check can take as long as two weeks to clear. Why is this so? As we saw in section 3.1, a given check may be forwarded by multiple banks before it reaches its original bank of issue. However, in *iCheck*, we can experience a vast increase in speed by moving the data from the analog domain to the digital domain. For example, without loss of generality, suppose a check in the analog domain takes 336 hours (2 weeks) to reach the original bank of issue, while a check in *iCheck* takes 1 hour to reach the Central Storage Module (a very conservative estimate); this is more than a 3000% decrease in check processing latency.

### **6.1.2 Drastic Decrease in Check Processing Cost.**

Two factors are responsible for the decrease in check processing costs that *iCheck* provides: OCR technology and digital check transportation. By performing OCR on the check, we eliminate the costs associated with employing a person to read the checks. For example, a person employed to read checks at \$5.00/hour for 8 hours/day working 250 days/year costs \$10000/year (and this is the cost for only one employee). This clearly surpasses any one-time cost associated with the *iCheck* OCR. Also, a consequence, which needs no explanation, of scanning checks into the digital domain is a decrease in check transportation costs.

### **6.1.3 Secure Transactions**

The heart of *iCheck*'s security protocols are based on the cryptographically safe algorithms RSA and RC4. Breaking RSA would imply solving an NP-complete problem in polynomial time which is believed by many mathematicians to be impossible [10]. Furthermore, RC4 has been scrutinized under conditions of non-disclosure by independent analysts and it is considered secure [24].

## **6.2 Future Work**

Future work to be done on *iCheck* includes improving the Optical Character Recognition (OCR) module to get higher recognition accuracy. Also, making the system scalable to handle thousands of bank modules by having multi-threaded support. Furthermore, an in-depth study of the economical, technological, and commercial repercussions that *iCheck* can potentially cause on the banking system.

### 6.3 How Can Other Systems Benefit from *iCheck*?

*iCheck* was designed in the client/server paradigm. Therefore, there is a clear demarcation of the workload distribution that other client/server systems can benefit from. Furthermore, *iCheck's* secure transactions are general enough to encompass the encryption of not only check images, but of any other potential data. Finally, other electronic currency systems can use the infra-structure and the cryptographical protocols in *iCheck* as ground work for their projects.

In summary, *iCheck* is an architecture that makes use of the existing financial and WWW infra-structure. It requires no special hardware. It automates and speeds up costly procedures such as reading the courtesy amount and check forwarding in a secure fashion. The technological advancements that the *iCheck* system exploits will enable banks and corporations to streamline and re-engineer back-office processes to provide faster check information, allowing check payment decisions to be made more quickly and with increased assurance. As a result, consumers, as well as businesses, should enjoy quicker availability of funds, better safeguards against fraud, and, in the near future, have greater and faster access to their checks' status at a customer site.

## Appendix A. Parser

```
#define _WIN32_WINNT 0x0400
#include <windows.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <io.h>

#define CABSIZ 50
#define FILENAMESIZ 40
#define PATHNAMESIZ 200
#define GREYMAPSIZ 1512038 /* 38 bytes of PGM header (1237538)*/
#define BRIEFCASEIDSIZ 20
#define BATCOMMNANDSIZ 50

#define MAXPATH 200
#define MAXENTRIES (40) /* Max number of name/value pairs */
#define MAXNAME (20) /* Max number of chars in each name + 1 */
#define MAXVALUE (100) /* Max number of chars in each value + 1 */

typedef char NameStr[MAXNAME];
typedef char ValueStr[MAXVALUE];

int read_str( FILE *f, char *s, int size )
{
    char ch;

    do { /* Read characters into s until null byte is read */
        ch = fgetc(f);
        if( ch == EOF )
            return -1;
        *s = (char) ch;
        s++;
        size--;
    } while( ch != '\0' && size != 0 );

    if( size == 0 ) /* Ensure string is null-terminated */
        *(s-1) = '\0';

    return 0;
}

int read_info( char *fn, NameStr n[], ValueStr v[] )
{
    int ix; /* Name/Value pairs read */
    FILE *f; /* File pointer to info file */

    f = fopen( fn, "rb" ); /* Open info file */
    if( f == NULL ) { /* return -1 on file open error */
        return -1;
    }
}
```

```

}
for( ix = 0; ix < MAXENTRIES; ix++ ) { /* Process all name/value pairs */
if( read_str( f, n[ix], MAXNAME ) == -1 ) /* Read name */
break; /* Endloop if end of file */
if( read_str( f, v[ix], MAXVALUE ) == -1 ) /* Read value */
break; /* Endwhile if end of file */
}
fclose( f ); /* Close file */

unlink( fn ); /* Delete info file */
return ix; /* Return number of entries read */
}

int main( int argc, char *argv[] )
{
FILE *fpUploaded, *fpimage, *fpHtmlfname;
char strCAB[CABSIZ], strImagefnamepath[PATHNAMESIZ];
char strImagefname[FILENAMESIZ], strHtmlfname[PATHNAMESIZ];
char byte;
char strheadhtml[] = "<HEADER>\n<TITLE> NETBANK </TITLE>\n<HEADER>\n<BODY><H1>
The image data</H1>\n";
char strBriefcaseID[BRIEFCASEIDSIZ], strBatCommand[BATCOMMNANDSIZ];
int i=0, ibytecount;

FILE *textfile; /* File pointer to textfile */
char infopath[MAXPATH]; /* Pathname of infofile */
char textpath[MAXPATH]; /* Pathname of textfile */
NameStr name[MAXENTRIES]; /* Names from info file */
ValueStr value[MAXENTRIES]; /* Values from info file */
int entries; /* # of Name/Value pairs from info file */
int ix; /* Index variable */

if( argc < 2 ) return 1; /* End program if argument is missing */

strcpy( infopath, argv[1] ); /* Get infofile path */

/* Read all info data */
entries = read_info( infopath, name, value );
if( entries < 0 ) return 1; /* End program if file error */

/***** Process the incoming data *****/

fpUploaded = fopen(value[1], "rb");

fscanf(fpUploaded,"%s",strBriefcaseID);//get ID for briefcase

while(fscanf(fpUploaded,"%s", strCAB) != EOF) {

sprintf(strImagefname, "check%d.pgm", i);
sprintf(strImagefnamepath,"c:\\inetpub\\wwwroot\\netbank\\conversions\\%s",strImagefname);
fpimage = fopen(strImagefnamepath, "w");

```

```

_setmode( _fileno( fpUploaded ), _O_BINARY );

fscanf(fpUploaded, "%c", &byte);

for(ibytecount=0; ibytecount<GREYMAPSIZE; ibytecount++) {
    fscanf(fpUploaded, "%c", &byte);
    fprintf(fpimage, "%c", byte);
}

sprintf(strBatCommand, "convert %s", strImagefname);
system(strBatCommand);
fclose(fpimage);
//_unlink(strImagefnamepath);//delete check%s.pgm file

//Create HTML file
sprintf(strHtmlfname, "c:\\inetpub\\wwwroot\\netbank\\conversions\\check%d.html",i);
fpHtmlfname = fopen(strHtmlfname,"w");
fprintf(fpHtmlfname, "%s", strheadhtml);
fprintf(fpHtmlfname, "<B>BriefCaseID = %s</B><P>\n\n", strBriefcaseID);
fprintf(fpHtmlfname, "<B>NumChk = %d</B><P>\n\n", i);
fprintf(fpHtmlfname, "<B>CAB = %s</B><P>\n\n", strCAB);
fprintf(fpHtmlfname, "<HR><IMG SRC='\"check%d.jpg'\"><P><HR></BODY></HTML>",i);
fclose(fpHtmlfname);
i++;
_setmode( _fileno( fpUploaded ), _O_TEXT );
}
fclose(fpUploaded);

strcpy(textpath, "userexec.txt");
textfile = fopen( textpath, "w" ); /* Open text file */
if( textfile == NULL ) { /* end program on file open error */
    fclose( textfile );
    return 1;
}

for( ix = 0; ix < entries; ix++ ) { /* Print all info to textfile */
    fprintf( textfile, "%s = %s\n", name[ix], value[ix] );
}

fclose( textfile ); /* Close textfile */

return 0; /* Done */
}

```

# Appendix B. Form Based File Uploading Sample HTML Page

```
<HEADER>
<TITLE> Netbank Home Page </TITLE>
<link href="mailto:joesurf@athena.mit.edu"></HEADER>

<BODY>

<H1> Netbank Home Page</H1>
<HR>

In the area below enter the name of the briefcase object to upload.<P>

<HR>
<form enctype="multipart/form-data"
      action="http://forrest.mit.edu/scripts/cgi-bin/upload.exe"
      method=post>
Enter Briefcase Object: <input name="filename" type="file"> <P>

<input type="submit" value=" Upload">
</form>
<HR>

</BODY>
</HTML>
```

## Appendix C. Certification Authority

```
#define _WIN32_WINNT 0x0400
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <winsock.h>
#include <winbase.h>
#include <wincrypt.h>
#include <iostream.h>
#include <tchar.h>

#define TEMPFILE "delme.txt"
#define BLOCK_SIZE 200
#define BUFFER_SIZE (BLOCK_SIZE+16) // Give buffer 16 bytes of extra
#define DEFAULT_PORT 1625
#define MAXLEN 512
#define HASH_SIZE 256
#define SDESCRIPTION "Hash of name and key"
#define BLOB_SIZE 256
#define HASH_SIZE 256

BOOL searchfor(int argc, char *argv[],
               char searchstring[2], int *whicharg)
{
    int count;
    // Display each command-line argument.
    for( count = 1; count < argc; count++ )
        if (memcmp(argv[count], searchstring, 2) == 0)
            {
                *whicharg = count;
                return TRUE;
            }
    return FALSE;
}

void main( int argc, // Number of strings in array argv
           char *argv[], // Array of command-line argument strings
           char *envp[] ) // Array of environment variable strings
{
    // repeat the loop?
    char repeatloop;

    //sockets declarations

    struct sockaddr_in serveraddr, clientaddr;
    int listensocket, connectedport, clientaddrlength, lettersent;

    //Cryptography declarations
    HCRYPTPROV hProv = 0;

    HCRYPTKEY hSignature = 0;
```

```

BYTE *pbSignature = NULL;
HCRYPTHASH hHash = 0;
DWORD dwSigLen;

//Winsock Version Verification
WORD wVersionRequested;
WSADATA wsaData;
int err;
int portnumber;
int scannedchar;

FILE *hSource = NULL;
FILE *hDest = NULL;
int eof = 0;
char filetowrite[NAME_SIZE];

int dwBlobBufLen, dwNameBufLen, OutBufLen;
BYTE pbBlobBuffer[MAXLEN];
BYTE nameBuffer[BUFFER_SIZE];
BYTE outBuffer[MAXLEN];
DWORD dwCount;
int whicharg;

// Read in port number
// port is defined with -p, DEFAULT otherwise
if (searchfor(argc, argv, "-p", &whicharg))
{
    portnumber = atoi(_strinc(argv[whicharg],2));
}
else
{
    portnumber = DEFAULT_PORT;
}

printf("Checking Winsock Versions.\n");

//Check versions of Winsock
wVersionRequested = MAKEWORD( 1, 1 );

err = WSASStartup( wVersionRequested, &wsaData );
if ( err != 0 ) {
    printf("Error %d during Startup\n", WSAGetLastError());

    /* Tell the user that we couldn't find a useable */
    /* WinSock DLL. */
    return;
}

/* Confirm that the WinSock DLL supports 2.0.*/
/* Note that if the DLL supports versions greater */
/* than 2.0 in addition to 2.0, it will still return */
/* 2.0 in wVersion since that is the version we */

```

```

/* requested.                */

if ( LOBYTE( wsaData.wVersion ) != 1 ||
    HIBYTE( wsaData.wVersion ) != 1 ) {
    printf("Could not find useable Winsock DLL\n");
    /* Tell the user that we couldn't find a useable */
    /* WinSock DLL.                */
    WSACleanup();
    return;
}

/* The WinSock DLL is acceptable. Proceed. */
listensocket = socket (AF_INET, SOCK_STREAM, 0);
if (listensocket < 0)
{
    printf("Error %d during create\n", WSAGetLastError());
    goto done;
}
printf("Socket created.\n");

// Assign listensocket to a local IP and port TEST_PORT
serveraddr.sin_family    = AF_INET;
serveraddr.sin_addr.s_addr = htonl (INADDR_ANY);
serveraddr.sin_port      = htons ((short)portnumber);
if (bind(listensocket, (struct sockaddr *) &serveraddr, sizeof (serveraddr))<0)
{
    printf("Error %d during bind.\n", WSAGetLastError());
    goto done;
}
printf("Socket bound.\n");

listen (listensocket, 5);

// Repeat the filename query and certificate signing in loop
do
{
    printf("Certificate authority ready and active on port %d\n", portnumber);

    // need to replace this "write to file idea"

    if (searchfor(argc, argv, "-j", &whicharg))
    {
        if((hDest=fopen(_strninc(argv[whicharg],2),"wb"))==NULL) {
            printf("Error opening Writing file!\n");
        }
        strcpy(filetowrite, (_strninc(argv[whicharg],2)));
    }
    else
    {
        if((hDest=fopen(TEMPFILE,"wb"))==NULL) {

```

```

    printf("Error opening Writing file!\n");
}
strcpy(filetowrite,TEMPFILE);
}

    // printf("Certification authority initialized.\nWaiting for connection.\n");

    clientaddrlen= sizeof(clientaddr);
    if ((connectedport = accept (listensocket,
(struct sockaddr *) &clientaddr,
&clientaddrlen)) < 0)
{
printf("Error %d during connection.\n", WSAGetLastError());
goto done;
}

    // printf("Connection successful. Receiving key.\n");

    if ((lettersent= recv (connectedport, pbBlobBuffer, BUFFER_SIZE, 0))<0)
{
printf("Error %d during receive.\n", WSAGetLastError());
goto done;
}

    // set dwBlobBufLen to size of public key blob received
    dwBlobBufLen= lettersent;

    if ((lettersent= recv (connectedport, nameBuffer, BUFFER_SIZE, 0))<0)
{
printf("Error %d during receive.\n", WSAGetLastError());
goto done;
}

    // set dwNameBufLen to size of name received
    dwNameBufLen = lettersent;

    // printf("Key and name received. Generating certificate.\n");

    // Get handle to the default provider.

    if(!CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_FULL, 0)) {
printf("Error %x during CryptAcquireContext!\n", GetLastError());
goto done;
}

    // Compute hash value and sign it.

    // Create hash object.
    if(!CryptCreateHash(hProv, CALG_MD5, 0, 0, &hHash)) {
printf("Error %x during CryptCreateHash (to send)!\n", GetLastError());
goto done;
}

    // Add Sender's Public Key to hash.

```

```

    if(!CryptHashData(hHash, pbBlobBuffer, dwBlobBufLen, 0)) {
printf("Error %x during CryptHashData (Name)!\n", GetLastError());
goto done;
    }

    // Add Sender's name to hash.
    if(!CryptHashData(hHash, nameBuffer, dwNameBufLen, 0)) {
printf("Error %x during CryptHashData (Name)!\n", GetLastError());
goto done;
    }

    // Get Signed Hash Length to allocate memory
    if(!CryptSignHash(hHash, AT_SIGNATURE, SDESCRIPTION, 0, NULL, &dwSigLen)) {
printf("Error %x during CryptGetHashParam!\n", GetLastError());
goto done;
    }

    if((pbSignature = malloc(dwSigLen)) == NULL) {
printf("Out of memory!\n");
goto done;
    }

    // Put Signed Hash in pbSignature
    if(!CryptSignHash(hHash, AT_SIGNATURE, SDESCRIPTION,
0, pbSignature, &dwSigLen)) {
printf("Error %x during CryptGetHashParam!\n", GetLastError());
goto done;
    }

    // Free up memory and destroy hashes
    if(!CryptDestroyHash(hHash)) {
printf("Error %x during CryptDestroyHash!\n", GetLastError());
goto done;
    }

    // Write size of signed hash to destination file.
    fwrite(&dwSigLen, sizeof(DWORD), 1, hDest);
    if(ferror(hDest)) {
printf("Error writing header!\n");
goto done;
    }

    // Keep track of outbuffer length
    OutBufLen=sizeof(DWORD);

    // Write size of name to destination file.
    fwrite(&dwNameBufLen, sizeof(DWORD), 1, hDest);
    if(ferror(hDest)) {
printf("Error writing header!\n");
goto done;
    }

    // Keep track of outbuffer length

```

```

OutBufLen=OutBufLen + sizeof(DWORD);

// Write size of public key to destination file.
fwrite(&dwBlobBufLen, sizeof(DWORD), 1, hDest);
if(ferror(hDest)) {
printf("Error writing header!\n");
goto done;
}

// Keep track of outbuffer length
OutBufLen=OutBufLen + sizeof(DWORD);

// Write signed hash to destination file.
fwrite(pbSignature, 1, dwSigLen, hDest);
if(ferror(hDest)) {
printf("Error writing Signature!\n");
free(pbSignature);
goto done;
}

// Keep track of outbuffer length
OutBufLen=OutBufLen + dwSigLen;

// Write sender name to destination file.
fwrite(nameBuffer, 1, dwNameBufLen, hDest);
if(ferror(hDest)) {
printf("Error writing Name!\n");
free(pbSignature);
goto done;
}

// Keep track of outbuffer length
OutBufLen=OutBufLen + dwNameBufLen;

fwrite(pbBlobBuffer, 1, dwBlobBufLen, hDest);
if(ferror(hDest)) {
printf("Error writing Public Key!\n");
goto done;
}

// Keep track of outbuffer length
OutBufLen=OutBufLen + dwBlobBufLen;

// Close destination file.
if(hDest != NULL) fclose(hDest);
// printf("Certificate written to file %s\n", filetowrite);

// Read file in again and transmit the certificate

if((hSource=fopen(filetowrite,"rb"))==NULL) {
printf("Error opening source file!\n");
}

dwCount = fread(outBuffer, 1, OutBufLen, hSource);

```

```

if(ferror(hSource)) {
    printf("Error reading data from source file!\n");
    goto done;
}
eof=feof(hSource);

// printf("Certificate Generated. Sending back.\n");

if ((lettersent= sendto (connectedport, outBuffer, OutBufLen, 0,(struct sockaddr *) &clientaddr, sizeof
(clientaddr)))<0)
{
    printf("Error %d during send.\n", WSAGetLastError());
    goto done;
}

// printf("Certificate sent.\n");

// Free memory used to store signature.
if(pbSignature != NULL) free(pbSignature);

// Continue?

closesocket(connectedport);

// Close destination file.
} while (1);
//while (strcmp(repeatloop,"y") || strcmp(repeatloop,"Y")==0);

done:

closesocket(listensocket);

}

```

## Appendix D.1. Three Phase Session Key Exchange Protocol (Server)

```
#define _WIN32_WINNT 0x0400
#include <windows.h>
#include <stdio.h>
#include <winsock.h>
#include <wincrypt.h>
#include <tchar.h>
#include <iostream.h> // for command line arguments

#define NAME_SIZE 256
#define BLOB_SIZE 512
#define BLOCK_SIZE 200
#define BUFFER_SIZE (BLOCK_SIZE+16) // Give buffer 16 bytes of extra

#define TEMPFILE "delme.txt"
#define DEFAULT_LOCAL_CERTFILE "mycert.txt"
#define DEFAULT_SIGNATURE_KEYFILE "sign.key"
#define SERVER_NAME "CSF"

#define NAME_LEN 60
#define HASH_SIZE 256

#define BUFFER_SIZE (BLOCK_SIZE+16) // Give buffer 16 bytes of extra
#define SDESCRIPTION "Hash of name and key"

BOOL verifycertificate(HCRYPTPROV hProv, char *filetocheck,
    char *signaturekeyfilename,
    BYTE *pbKeyBlob, DWORD *dwBlobLen,
    BYTE *pbDestUserName, DWORD *dwDestUserNameLen);
BOOL searchfor(int argc, char *argv[],
    char searchstring[2], int *whicharg);

// verify certificate returns 1 if certificate checks okay, 0 otherwise
// input hProv-HCRYPTPROV cryptography provider
// input filetocheck-char null-terminated string of certificate to verify
// output publickeyblob-public key blob of destination user
// output dwBlobLen-Length of public key blob
// output pbDestUserName-null-terminated string for destination user name
// output dwDestUserNameLen-Length of name

BOOL verifycertificate(HCRYPTPROV hProv, char *filetocheck,
    char *signaturekeyfilename,
    BYTE *pbKeyBlob, DWORD *dwBlobLen,
    BYTE *pbDestUserName, DWORD *dwDestUserNameLen){

FILE *hDest = NULL;
FILE *hSign = NULL;
```

```

BYTE pbSignKeyBlob[BLOB_SIZE];
DWORD dwSignBlobLen;
HCRYPTKEY hSignature = 0;
HCRYPTHASH hHash = 0;

DWORD PublicKeyLen, HashLen, NameLen;
BYTE FilePublicKey[BLOB_SIZE], FileHash[BLOB_SIZE], FileName[NAME_SIZE];

int scannedchar;

if((hSign=fopen(signaturekeyfilename,"rb"))==NULL) {
    printf("Error opening source file!\n");
}

// Read signature key blob length from source file and allocate memory.
fread(&dwSignBlobLen, sizeof(DWORD), 1, hSign);
if(ferror(hSign) || feof(hSign)) {
    printf("Error reading file header!\n");
    return(FALSE);
}

pbSignKeyBlob == malloc(dwSignBlobLen);
if(pbSignKeyBlob == NULL) {
    printf("Out of memory!\n");
    return(FALSE);
}

// Read signature key blob from source file.
fread(pbSignKeyBlob, 1, dwSignBlobLen, hSign);
if(ferror(hSign) || feof(hSign)) {
    printf("Error reading file header!\n");
    return(FALSE);
}

// Import signature key blob into CSP.
if(!CryptImportKey(hProv, pbSignKeyBlob, dwSignBlobLen, 0, 0, &hSignature)) {
    printf("Error %x during CryptImportKey!\n", GetLastError());
    return(FALSE);
}

// check certificate
printf("Verifying certificate.\n");

if((hDest=fopen(filetocheck,"rb"))==NULL) {
    printf("Error opening Certificate file for examination!\n");
    return(FALSE);
}

// Allocate memory for Hash
fread(&HashLen, sizeof(DWORD), 1, hDest);
if(ferror(hDest) || feof(hDest)) {
    printf("Error reading file header!\n");
    return(FALSE);
}

```

```

}

FileHash == malloc(HashLen);
if(FileHash == NULL) {
    printf("Out of memory!\n");
    return(FALSE);
}

// Allocate memory for Name
fread(&NameLen, sizeof(DWORD), 1, hDest);
if(ferror(hDest) || feof(hDest)) {
    printf("Error reading file header!\n");
    return(FALSE);
}

FileName == malloc(NameLen);
if(FileName == NULL) {
    printf("Out of memory!\n");
    return(FALSE);
}

// Allocate memory for Public Key Blob
fread(&PublicKeyLen, sizeof(DWORD), 1, hDest);
if(ferror(hDest) || feof(hDest)) {
    printf("Error reading Public Key!\n");
    return(FALSE);
}

FilePublicKey == malloc(PublicKeyLen);
if(FilePublicKey == NULL) {
    printf("Out of memory!\n");
    return(FALSE);
}

// Read signature
fread(FileHash, 1, HashLen, hDest);
if(ferror(hDest) || feof(hDest)) {
    printf("Error reading hash!\n");
    return(FALSE);
}

// Read name
fread(FileName, 1, NameLen, hDest);
if(ferror(hDest) || feof(hDest)) {
    printf("Error reading name!\n");
    return(FALSE);
}

// Read public key
fread(FilePublicKey, 1, PublicKeyLen, hDest);
if(ferror(hDest) || feof(hDest)) {
    printf("Error reading key!\n");
    return(FALSE);
}

```

```

// Create hash object.
if(!CryptCreateHash(hProv, CALG_MD5, 0, 0, &hHash)) {
    printf("Error %x during CryptCreateHash (to send)!\n", GetLastError());
    return(FALSE);
}

// Add Sender's Public Key to hash.
if(!CryptHashData(hHash, FilePublicKey, PublicKeyLen, 0)) {
    printf("Error %x during CryptHashData (Name)!\n", GetLastError());
    return(FALSE);
}

// Add Sender's name to hash.
if(!CryptHashData(hHash, FileName, NameLen, 0)) {
    printf("Error %x during CryptHashData (Name)!\n", GetLastError());
    return(FALSE);
}

if(!CryptVerifySignature(hHash, FileHash, HashLen, hSignature, SDESCRIPTION, 0)) {
    printf("Error %x during CryptVerifySignature!\n", GetLastError());
    printf("Certificate invalid!\n");
    return(FALSE);
    //delete file
}
else
{
    // printf("Certificate %s verified and ready to use!\n",filetocheck);
    memcpy(pbKeyBlob,FilePublicKey,PublicKeyLen);
    *dwBlobLen=PublicKeyLen;
    memcpy(pbDestUserName,FileName,NameLen);
    //pbDestUserName= FileName
    *dwDestUserNameLen=NameLen;

    if(hDest != NULL) fclose(hDest);
    if(hSign != NULL) fclose(hSign);

    return(TRUE);
}
}

```

```

// searchfor finds a switch in the command line and returns the
// argument number of the switch
// searchfor returns true if the searchstring is found
// argc (int) = number of command line items (excluding call)
// argv (*char) = string values of these command line items
// searchstring[2] (char) = switch to be searched for i.e. "-g"
// whicharg (int) = returns the value of the found arg

```

```

BOOL searchfor(int argc, char *argv[],
    char searchstring[2], int *whicharg)

```

```

{
    int count;
    // Display each command-line argument.
    for( count = 1; count < argc; count++ )
        if (memcmp(argv[count], searchstring, 2) == 0)
            {
                *whicharg = count;
                return TRUE;
            }
    return FALSE;
}

void main(int argc,    // Number of strings in array argv
          char *argv[], // Array of command-line argument strings
          char *envp[] ) // Array of environment variable strings
{
    //sockets declarations
#define DEFAULT_PORT 1625
#define MAXLEN 1000
#define HASH_SIZE 256
#define CERT_SIZE 512
#define BUFFER_SIZE (BLOCK_SIZE+16) // Give buffer 16 bytes of extra

    struct sockaddr_in serveraddr, clientaddr;
    int listensocket, connectedport, clientaddrlen, lettersent;

    //Winsock Version Verification
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;

    char signaturekeyfilename[60];

    //Cryptography declarations
    HCRYPTPROV hProv = 0;
    BYTE pbDestName[NAME_SIZE];
    //char pbDestName[NAME_SIZE];
    DWORD dwDestNameLen;
    BYTE pbSendName[NAME_SIZE];
    //char pbSendName[NAME_SIZE];
    DWORD dwSendNameLen;
    HCRYPTKEY hSendPubKey = 0;
    HCRYPTKEY hMyPubKey = 0;
    HCRYPTKEY hSharedKey = 0;
    HCRYPTKEY hKeyA = 0;
    HCRYPTKEY hKeyB = 0;
    BYTE pbKeyBlob[BLOB_SIZE];
    DWORD dwBlobLen;

    BYTE pbHash[HASH_SIZE];
    DWORD dwHashLen;
    BYTE pbSendHash[HASH_SIZE];
    DWORD dwSendHashLen;

```

```

HCRYPTHASH hHash = 0;

BYTE pbCert[CERT_SIZE];
DWORD Certlength;
DWORD Certpartlength;
BYTE pboutbuffer[MAXLEN];

//File access variables
FILE *hSource = NULL;
FILE *hRemoteCert = NULL;
FILE *hDest = NULL;
FILE *hRecvFile = NULL;
int eof = 0;

// room for padding, etc.
BYTE pbBuffer[BUFFER_SIZE];
DWORD dwCount;

char localcertfilename[60], keyfilename[60], remotecertfilename[60];
char RecvBuffer[BUFFER_SIZE], recvfilename[60];
int scannedchar;
int portnumber;
int whicharg;
LONG ReceiveLen;
LONG BytesReceived;

// Read in client (local) port number
// client port is defined with -p, DEFAULT otherwise
if (searchfor(argc, argv, "-p", &whicharg))
{
    portnumber = atoi(_strninc(argv[whicharg],2));
}
else
{
    portnumber = DEFAULT_PORT;
}

// Read in LocalCertificate
// local certificate is defined with -m, DEFAULT otherwise
if (searchfor(argc, argv, "-m", &whicharg))
{
    if((hSource=fopen(_strninc(argv[whicharg],2),"rb"))==NULL) {
printf("Error opening your certificate file!\n");
goto done;
    }
    strcpy(localcertfilename, (_strninc(argv[whicharg],2)));
}

else
{
    if((hSource=fopen(DEFAULT_LOCAL_CERTFILE,"rb"))==NULL) {
printf("Error opening your certificate file!\n");
goto done;
    }
}

```

```

    strcpy(localcertfilename, DEFAULT_LOCAL_CERTFILE);
}

// Read in junk filename
if (searchfor(argc, argv, "-j", &whicharg))
{
    if((hRemoteCert=fopen((_strninc(argv[whicharg],2)),"wb"))==NULL) {
printf("Error opening Writing file!\n");
    }
    strcpy(remotecertfilename, (_strninc(argv[whicharg],2)));
}
else

{
    if((hRemoteCert=fopen(TEMPFILE,"wb"))==NULL) {
printf("Error opening certificate destination file!\n");
    }
    strcpy(remotecertfilename,TEMPFILE);
}

// Get handle to the default provider.

if(!CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_FULL, 0)) {
    printf("Error %x during CryptAcquireContext!\n", GetLastError());
    goto done;
}

//Initiate connection

//Check versions of Winsock
wVersionRequested = MAKEWORD( 1, 1 );

err = WSAStartup( wVersionRequested, &wsaData );
if ( err != 0 ) {
    printf("Error %d during Startup\n", WSAGetLastError());

    /* Tell the user that we couldn't find a useable */
    /* WinSock DLL. */
    return;
}

/* Confirm that the WinSock DLL supports 2.0.*/
/* Note that if the DLL supports versions greater */
/* than 2.0 in addition to 2.0, it will still return */
/* 2.0 in wVersion since that is the version we */
/* requested. */

if ( LOBYTE( wsaData.wVersion ) != 1 ||
    HIBYTE( wsaData.wVersion ) != 1 ) {
    printf("Could not find useable Winsock DLL\n");
    /* Tell the user that we couldn't find a useable */
    /* WinSock DLL. */
    WSACleanup( );
    return;
}

```

```

}

/* The WinSock DLL is acceptable. Proceed. */

listensocket = socket (AF_INET, SOCK_STREAM, 0);
if (listensocket < 0)
{
    printf("Error %d during create\n", WSAGetLastError());
    goto done;
}
//printf("Socket created.\n");

// Assign listensocket to a local IP and port TEST_PORT
serveraddr.sin_family = AF_INET;
serveraddr.sin_addr.s_addr = htonl (INADDR_ANY);
serveraddr.sin_port = htons ((short)portnumber);
if (bind(listensocket, (struct sockaddr *) &serveraddr, sizeof (serveraddr))<0)
{
    printf("Error %d during bind.\n", WSAGetLastError());
    goto done;
}
listen (listensocket, 5);

//repeat this whole loop infinitely

do{

    printf("Server active on port %d\n", portnumber);

    clientaddrlength= sizeof(clientaddr);
    if ((connectedport = accept (listensocket, (struct sockaddr *) &clientaddr, &clientaddrlength)) < 0)
    {
        printf("Error %d during connection.\n", WSAGetLastError());
        goto done;
    }

    printf("Connection successful.\n");

    // Receive a key blob containing certificate

    //printf("Receiving remote Certificate\n");

    if ((lettersent= recv (connectedport, pbCert, CERT_SIZE, 0))<0)
    {
        printf("Error %d during receive of certificate.\n", WSAGetLastError());
        goto done;
    }

    // Write data to destination file.
    fwrite(pbCert, 1, lettersent, hRemoteCert);
    if(ferror(hRemoteCert)) {
        printf("Error writing data to remote certificate file!\n");
        goto done;
    }
}

```

```

if(hRemoteCert != NULL) fclose(hRemoteCert);

// Obtain the sending user's exchange public key. Import it into the
// CSP and place a handle to it in 'hSendPubKey'.

// printf("Sending my certificate.\n");

// Get certificate length from file header
fread(&Certpartlength, sizeof(DWORD), 1, hSource);
Certlength=Certpartlength+sizeof(DWORD);
fread(&Certpartlength, sizeof(DWORD), 1, hSource);
Certlength=Certpartlength+Certlength+sizeof(DWORD);
fread(&Certpartlength, sizeof(DWORD), 1, hSource);
Certlength=Certpartlength+Certlength+sizeof(DWORD);
//rewind to start and read again
rewind(hSource);

dwCount = fread(pboutbuffer, 1, Certlength, hSource);
if(ferror(hSource)) {
    printf("Error reading data from source file!\n");
    goto done;
}
eof=feof(hSource);

if ((lettersent= sendto (connectedport, pboutbuffer, Certlength, 0,(struct sockaddr *) &clientaddr, sizeof
(clientaddr)))<0)

{
printf("Error %d during send.\n", WSAGetLastError());
goto done;
}

//Signature key filename -S
if (searchfor(argc, argv, "-S", &whicharg))
{
strcpy(signaturekeyfilename, (_strninc(argv[whicharg],2)));
}
else
{
strcpy(signaturekeyfilename, DEFAULT_SIGNATURE_KEYFILE);
}

if (!verifycertificate(hProv, remotecertfilename, signaturekeyfilename,
pbKeyBlob, &dwBlobLen,
pbDestName, &dwDestNameLen))
{
printf("Could not verify certificate.\n");
goto done;
}

// Import key blob into CSP.
if(!CryptImportKey(hProv, pbKeyBlob, dwBlobLen, 0, 0, &hSendPubKey)) {

```

```

    printf("Error %x during CryptImportKey!\n", GetLastError());
    goto done;
}

//Do I need this line? I'm overwriting the DestPubKey

if(!CryptGetUserKey(hProv, AT_KEYEXCHANGE, &hMyPubKey)) {
    printf("Error %x during CryptGetUserKey\n", GetLastError());
    goto done;
}

// Obtain the sending user's name. This is usually done at the
// same time the public key was obtained. Place this in
// 'pbSendName' and set 'dwSendNameLen' to the number of bytes in
// the name.

// Hardcode the server's name as "CSF"
strcpy(pbSendName,SERVER_NAME);
dwSendNameLen=strlen(pbSendName);

// Receive a key blob containing session key A from the sending user
// and place it in 'pbKeyBlob'. Set 'dwBlobLen' to the number of
// bytes in the key blob.

// printf("Receiving Session Key A.\n");

if ((lettersent= recv (connectedport, pbKeyBlob, MAXLEN, 0))<0)
{
printf("Error %d during receive.\n", WSAGetLastError());
goto done;
}
dwBlobLen=lettersent;

// Import the key blob into the CSP.
if(!CryptImportKey(hProv, pbKeyBlob, dwBlobLen, 0, 0, &hKeyA)) {
    printf("Error %x during CryptImportKey KeyA!\n", GetLastError());
    goto done;
}

// Create a random session key (session key B). Because this key is
// going to be used solely for key exchange and not encryption, it
// does not matter which algorithm you specify here.
if(!CryptGenKey(hProv, CALG_RC2, CRYPT_EXPORTABLE, &hKeyB)) {
    printf("Error %x during CryptGenKey KeyB!\n", GetLastError());
    goto done;
}

// Export session key B into a simple key blob.
dwBlobLen = BLOB_SIZE;

if(!CryptExportKey(hKeyB, hSendPubKey,

```

```

    SIMPLEBLOB, 0, pbKeyBlob, &dwBlobLen)) {
    printf("Error %x during CryptExportKey KeyB!\n", GetLastError());
    goto done;
}

// Transmit key blob containing session key B to the sending user.

//printf("Sending Session Key B.\n");

if ((lettersent= sendto (connectedport, pbKeyBlob,
    dwBlobLen, 0,(struct sockaddr *) &clientaddr,
    sizeof (clientaddr))<0)
    {
    printf("Error %d during send.\n", WSAGetLastError());
    goto done;
    }

//
// Compute hash value and transmit it to the sending user.
//
// Create hash object.
if(!CryptCreateHash(hProv, CALG_MD5, 0, 0, &hHash)) {
    printf("Error %x during CryptCreateHash (to send)!\n", GetLastError());
    goto done;
}

// Add session key A to hash.
if(!CryptHashSessionKey(hHash, hKeyA, 0)) {
    printf("Error %x during CryptHashSessionKey (KeyA)!\n", GetLastError());
    goto done;
}

// Add destination user's name to hash.

if(!CryptHashData(hHash, pbDestName, dwDestNameLen, 0)) {
    printf("Error %x during CryptHashData (DestName)!\n", GetLastError());
    goto done;
}

// Add session key B to hash.

if(!CryptHashSessionKey(hHash, hKeyB, 0)) {
    printf("Error %x during CryptHashSessionKey (keyB)!\n", GetLastError());
    goto done;
}

// Add sending user name to hash.

if(!CryptHashData(hHash, pbSendName, dwSendNameLen, 0)) {
    printf("Error %x during CryptHashData (SendName)!\n", GetLastError());
    goto done;
}

// Add "phase 2" text to hash.

```

```

if(!CryptHashData(hHash, "phase 2", 7, 0)) {
    printf("Error %x during CryptHashData (phase2)!\n", GetLastError());
    goto done;
}

// Complete the hash computation and retrieve the hash value.
dwHashLen = HASH_SIZE;

if(!CryptGetHashParam(hHash, HP_HASHVAL, pbHash, &dwHashLen, 0)) {
    printf("Error %x during CryptGetHashParam!\n", GetLastError());
    goto done;
}

// Destroy the hash object.

if(!CryptDestroyHash(hHash)) {
    printf("Error %x during CryptDestroyHash!\n", GetLastError());
    goto done;
}

// Transmit the hash value to the sending user.

if ((lettersent= sendto (connectedport, pbHash, dwHashLen, 0,(struct sockaddr *) &clientaddr, sizeof (cli-
entaddr)))<0)
    {
    printf("Error %d during send.\n", WSAGetLastError());
    goto done;
    }

// Wait for the sending user to respond.

// Receive a hash value from the sending user and place it in
// 'pbSendHashValue'. Set 'dwSendHashLen' to the number of bytes in
// the hash value.

//printf("Receiving 2nd hash value.\n");

if ((lettersent= recv (connectedport, pbSendHash, MAXLEN, 0)) < 0)
    {
    printf("Error %d during recv.\n", WSAGetLastError());
    goto done;
    }
dwSendHashLen=lettersent;

// Verify hash value received from the sending user.

// Create hash object.
if(!CryptCreateHash(hProv, CALG_MD5, 0, 0, &hHash)) {
    printf("Error %x during CryptCreateHash (to compare)!\n",
    GetLastError());
    goto done;
}

```

```

// Add session key B to hash.
if(!CryptHashSessionKey(hHash, hKeyB, 0)) {
    printf("Error %x during CryptHashSessionKey (hKeyB)!\n", GetLastError());
    goto done;
}

// Add sending user's name to hash.
if(!CryptHashData(hHash, pbSendName, dwSendNameLen, 0)) {
    printf("Error %x during CryptHashData (sendName)!\n", GetLastError());
    goto done;
}

// Add destination user's name to hash.
if(!CryptHashData(hHash, pbDestName, dwDestNameLen, 0)) {
    printf("Error %x during CryptHashData (destName)!\n", GetLastError());
    goto done;
}

// Add "phase 3" text to hash.
if(!CryptHashData(hHash, "phase 3", 7, 0)) {
    printf("Error %x during CryptHashData (phase 3)!\n", GetLastError());
    goto done;
}

// Complete the hash computation and retrieve the hash value.
dwHashLen = HASH_SIZE;
if(!CryptGetHashParam(hHash, HP_HASHVAL, pbHash, &dwHashLen, 0)) {
    printf("Error %x during CryptGetHashParam!\n", GetLastError());
    goto done;
}

// Destroy the hash object.
if(!CryptDestroyHash(hHash)) {
    printf("Error %x during CryptDestroyHash!\n", GetLastError());
    goto done;
}

//
// Compare the hash value received from the sending user with the
// hash value that we just computed. If they do not match, then
// terminate the protocol.
//
if(dwHashLen!=dwSendHashLen || memcmp(pbHash, pbSendHash, dwHashLen)) {
    printf("Key exchange protocol failed in phase 3!\n");
    printf("Aborting protocol!\n");
    return;
}

//printf("Hash validated.\n");

//
// Create a shared session key to be used by the two users for
// exchanging encrypted messages. Both users must agree on the
// algorithm and parameters that this key is to use.

```

```

//
// Create hash object.

if(!CryptCreateHash(hProv, CALG_MD5, 0, 0, &hHash)) {
    printf("Error %x during CryptCreateHash (shared key)\n",
        GetLastError());
    goto done;
}

// Add session key A to hash.

if(!CryptHashSessionKey(hHash, hKeyA, 0)) {
    printf("Error %x during CryptHashSessionKey (KeyA)\n", GetLastError());
    goto done;
}

// Add session key B to hash.

if(!CryptHashSessionKey(hHash, hKeyB, 0)) {
    printf("Error %x during CryptHashSessionKey (KeyB)\n", GetLastError());
    goto done;
}
// Complete the hash computation and derive a session key from it.
// The CRYPT_EXPORTABLE flag is not specified here because the key
// is not generally exported out of the CSP.

//Modified to CRYPT_EXPORTABLE from 0

if(!CryptDeriveKey(hProv, CALG_RC4, hHash, CRYPT_EXPORTABLE, &hSharedKey)) {
    printf("Error %x during CryptDeriveKey!\n", GetLastError());
    goto done;
}
// Destroy the hash object.

if(!CryptDestroyHash(hHash)) {
    printf("Error %x during CryptDestroyHash!\n", GetLastError());
    goto done;
}

// Use the shared key to send encrypted messages to the other user.

/* send stuff */
printf("Successful Key Exchange.\n");

// get file length
if ((lettersent= recv (connectedport, RecvBuffer, 20, 0))<0)
    {
printf("Error %d during receive.\n", WSAGetLastError());
goto done;
    }

if ((lettersent= sendto (connectedport, RecvBuffer, strlen(RecvBuffer),
0,(struct sockaddr *) &clientaddr,
sizeof (clientaddr)))<0)

```

```

    {
printf("Error %d during send.\n", WSAGetLastError());
goto done;
    }

    ReceiveLen=atol(RecvBuffer);

    // get file name
    if ((lettersent= recv (connectedport, RecvBuffer, 20, 0))<0)
    {
printf("Error %d during receive.\n", WSAGetLastError());
goto done;
    }

    if ((lettersent= sendto (connectedport, RecvBuffer, strlen(recvfilename),
    0,(struct sockaddr *) &clientaddr,
    sizeof (clientaddr)))<0)
    {
printf("Error %d during send.\n", WSAGetLastError());
goto done;
    }

    strncpy(recvfilename, RecvBuffer, lettersent);

    //open file
    if((hRecvFile=fopen(recvfilename,"wb"))==NULL) {
        printf("Error opening key destination file!\n");
    }

    // receive file
    eof = 0;
    BytesReceived =0;
    do {
        if ((lettersent= recv (connectedport, RecvBuffer, BUFFER_SIZE, 0))<0)
        {
printf("Error %d during receive.\n", WSAGetLastError());
goto done;
        }

        // decrypt data
        if(!((BytesReceived +lettersent)<ReceiveLen))
        {
eof = 1;
        }

        if(!CryptDecrypt(hSharedKey, 0, eof, 0, RecvBuffer, &lettersent)) {
            printf("Error %x during CryptDecrypt!\n", GetLastError());
            goto done;
        }
        fwrite(RecvBuffer, 1, lettersent, hRecvFile);
        if(ferror(hRecvFile)) {
            printf("Error writing data to destination file!\n");
            goto done;
        }
    }

```

```

    BytesReceived = BytesReceived + lettersent;
} while(BytesReceived < ReceiveLen );

// Close source file.
if(hSource != NULL) fclose(hSource);

// Close destination file.
if(hDest != NULL) fclose(hDest);

if(hRecvFile != NULL) fclose(hRecvFile);

} while(1); //repeat this large loop

done:

// Destroy session key.
// Destroy handle to sending user's public key.
if(!CryptDestroyKey(hSharedKey)) {
    printf("Error %x during CryptDestroyKey!\n", GetLastError());
    exit (2);
}
// Release provider handle.

if(!CryptReleaseContext(hProv, 0)) {
    printf("Error %x during CryptReleaseContext!\n", GetLastError());
    exit (2);
}

// Close source file.
if(hSource != NULL) fclose(hSource);

// Close destination file.
if(hDest != NULL) fclose(hDest);

if (closesocket(connectedport) < 0)
{
    printf("Error %d during closesocket.\n", WSAGetLastError());
    exit (3);
}

if (closesocket(listensocket) < 0)
{
    printf("Error %d during closesocket.\n", WSAGetLastError());
    exit (3);
}
}

```

## Appendix D.2. Three Phase Session Key Exchange Protocol (Client)

```
#define _WIN32_WINNT 0x0400
#include <windows.h>
#include <stdio.h>
#include <winsock.h>
#include <wincrypt.h>
#include <tchar.h>
#include <iostream.h> // for command line arguments

#define NAME_SIZE 256
#define BLOB_SIZE 512
#define BLOCK_SIZE 200

#define TEMPFILE "delme.txt"
#define DEFAULT_LOCAL_CERTFILE "mycert.txt"
#define DEFAULT_SIGNATURE_KEYFILE "sign.key"
#define DEFAULT_SERVERIP "18.172.0.225"

#define NAME_LEN 60
#define HASH_SIZE 256

// Prototypes
BOOL verifycertificate(HCRYPTPROV hProv, char *filetocheck, char *signaturekeyfilename,
    BYTE *pbKeyBlob, DWORD *dwBlobLen,
    BYTE *pbDestUserName, DWORD *dwDestUserNameLen);
BOOL askforserverip(char *serveripaddress);
BOOL searchfor(int argc, char *argv[],
    char searchstring[2], int *whicharg);

// searchfor finds a switch in the command line and returns the
// argument number of the switch
// searchfor returns true if the searchstring is found
// argc (int) = number of command line items (excluding call)
// argv (*char) = string values of these command line items
// searchstring[2] (char) = switch to be searched for i.e. "-g"
// whicharg (int) = returns the value of the found arg

BOOL searchfor(int argc, char *argv[],
    char searchstring[2], int *whicharg)
{
    int count;
    // Display each command-line argument.
    for( count = 1; count < argc; count++ )
        if (memcmp(argv[count], searchstring, 2) == 0)
        {
            *whicharg = count;
            return TRUE;
        }
    return FALSE;
}
```

```

BOOL askforserverip(char *serveripaddress){

    char *dotfinder, *testipaddress;
    int dotcount, found3dots;
    int scannedchar;

    // Read in server IP address

    dotcount = 0;
    found3dots = 0;
    testipaddress=serveripaddress;
    while (dotcount < 3){
        dotfinder = strchr(testipaddress, '.');
        if (dotfinder == NULL){
            found3dots = 1;
            dotcount = 3;
        }
        testipaddress = _strinc(dotfinder);
        dotcount++;
    }
    if (found3dots == 1){
        printf("IP address not formatted correctly.\n");
        return FALSE;
    }
    else
    {
        return TRUE;
    }
}

// verifycertificate returns 1 if certificate checks okay, 0 otherwise
// input hProv-HCRYPTPROV cryptography provider
// input filetocheck-char null-terminated string of certificate to verify
// output publickeyblob-public key blob of destination user
// output dwBlobLen-Length of public key blob
// output pbDestUserName-null-terminated string for destination user name
// output dwDestUserNameLen-Length of name
BOOL verifycertificate(HCRYPTPROV hProv,char *filetocheck, char *signaturekeyfilename,
    BYTE *pbKeyBlob, DWORD *dwBlobLen,
    BYTE *pbDestUserName, DWORD *dwDestUserNameLen){

    FILE *hDest = NULL;
    FILE *hSign = NULL;

#define BUFFER_SIZE (BLOCK_SIZE+16) // Give buffer 16 bytes of extra
#define SDESCRIPTION "Hash of name and key"

    BYTE pbSignKeyBlob[BLOB_SIZE];
    DWORD dwSignBlobLen;
    HCRYPTKEY hSignature = 0;
    HCRYPTHASH hHash = 0;

```

```

DWORD PublicKeyLen, HashLen, NameLen;
BYTE FilePublicKey[BLOB_SIZE], FileHash[BLOB_SIZE], FileName[NAME_SIZE];

//char signaturekeyfilename[NAME_LEN];
int scannedchar;
int whicharg;

do{
    // Open source file.
    if((hSign=fopen(signaturekeyfilename,"rb"))==NULL) {
        printf("Error opening source file!\n");
    }
} while (hSign==NULL);

// Read signature key blob length from source file and allocate memory.
fread(&dwSignBlobLen, sizeof(DWORD), 1, hSign);
if(ferror(hSign) || feof(hSign)) {
    printf("Error reading signature length!\n");
    return(FALSE);
}

pbSignKeyBlob == malloc(dwSignBlobLen);
if(pbSignKeyBlob == NULL) {
    printf("Out of memory!\n");
    return(FALSE);
}

// Read signature key blob from source file.
fread(pbSignKeyBlob, 1, dwSignBlobLen, hSign);
if(ferror(hSign) || feof(hSign)) {
    printf("Error reading signature blob!\n");
    return(FALSE);
}

// Import signature key blob into CSP.
if(!CryptImportKey(hProv, pbSignKeyBlob, dwSignBlobLen, 0, 0, &hSignature)) {
    printf("Error %x during CryptImportKey!\n", GetLastError());
    return(FALSE);
}

// check certificate
printf("Verifying certificate.\n");

if((hDest=fopen(filetocheck,"rb"))==NULL) {
    printf("Error opening Certificate file for examination!\n");
    return(FALSE);
}

// Allocate memory for Hash
fread(&HashLen, sizeof(DWORD), 1, hDest);
if(ferror(hDest) || feof(hDest)) {
    printf("Error reading file header!\n");
    return(FALSE);
}

```

```

}

FileHash == malloc(HashLen);
if(FileHash == NULL) {
    printf("Out of memory!\n");
    return(FALSE);
}

// Allocate memory for Name
fread(&NameLen, sizeof(DWORD), 1, hDest);
if(ferror(hDest) || feof(hDest)) {
    printf("Error reading file header!\n");
    return(FALSE);
}

FileName == malloc(NameLen);
if(FileName == NULL) {
    printf("Out of memory!\n");
    return(FALSE);
}

// Allocate memory for Public Key Blob
fread(&PublicKeyLen, sizeof(DWORD), 1, hDest);
if(ferror(hDest) || feof(hDest)) {
    printf("Error reading Public Key!\n");
    return(FALSE);
}

FilePublicKey == malloc(PublicKeyLen);
if(FilePublicKey == NULL) {
    printf("Out of memory!\n");
    return(FALSE);
}

// Read signature
fread(FileHash, 1, HashLen, hDest);
if(ferror(hDest) || feof(hDest)) {
    printf("Error reading hash!\n");
    return(FALSE);
}

// Read name
fread(FileName, 1, NameLen, hDest);
if(ferror(hDest) || feof(hDest)) {
    printf("Error reading name!\n");
    return(FALSE);
}

// Read public key
fread(FilePublicKey, 1, PublicKeyLen, hDest);
if(ferror(hDest) || feof(hDest)) {
    printf("Error reading key!\n");
    return(FALSE);
}

```

```

// Create hash object.
if(!CryptCreateHash(hProv, CALG_MD5, 0, 0, &hHash)) {
    printf("Error %x during CryptCreateHash (to send)!\n", GetLastError());
    return(FALSE);
}

// Add Sender's Public Key to hash.
if(!CryptHashData(hHash, FilePublicKey, PublicKeyLen, 0)) {
    printf("Error %x during CryptHashData (Name)!\n", GetLastError());
    return(FALSE);
}

// Add Sender's name to hash.
if(!CryptHashData(hHash, FileName, NameLen, 0)) {
    printf("Error %x during CryptHashData (Name)!\n", GetLastError());
    return(FALSE);
}

if(!CryptVerifySignature(hHash, FileHash, HashLen, hSignature, SDESCRIPTION, 0)) {
    printf("Error %x during CryptVerifySignature!\n", GetLastError());
    printf("Certificate invalid\n");
    return(FALSE);
    //delete file
}
else
{
    printf("Certificate %s verified and ready to use\n",filetocheck);
    memcpy(pbKeyBlob,FilePublicKey,PublicKeyLen);
    *dwBlobLen=PublicKeyLen;
    memcpy(pbDestUserName,FileName,NameLen);
    *dwDestUserNameLen=NameLen;
    if(hDest != NULL) fclose(hDest);
    if(hSign != NULL) fclose(hSign);
    return(TRUE);
}
}

void main(int argc,    // Number of strings in array argv
          char *argv[], // Array of command-line argument strings
          char *envp[] ) // Array of environment variable strings
{
    //sockets declarations
#define DEFAULT_SERVER_PORT 1625
#define DEFAULT_CLIENT_PORT 1626
#define MAXLEN 1000
#define NAME_SIZE 256
#define CERT_SIZE 512
#define HASH_SIZE 256
#define BUFFER_SIZE (BLOCK_SIZE+16) // Give buffer 16 bytes of extra
#define IP_ADDRESS_LEN 20

    struct sockaddr_in serveraddr, clientaddr;

```

```

int connectedport, connecterror, lettersent;
int localportnumber, connectportnumber;

BYTE pboutbuffer[MAXLEN];

//Winsock Version Verification
WORD wVersionRequested;
WSADATA wsaData;
int err;

//Cryptography declarations
HCRYPTPROV hProv = 0;

BYTE pbDestName[NAME_SIZE];
//char pbDestName[NAME_SIZE];
DWORD dwDestNameLen;
BYTE pbSendName[NAME_SIZE];
//char pbSendName[NAME_SIZE];
DWORD dwSendNameLen;
HCRYPTKEY hDestPubKey = 0;
HCRYPTKEY hMyPubKey = 0;
HCRYPTKEY hSharedKey = 0;
HCRYPTKEY hKeyA = 0;
HCRYPTKEY hKeyB = 0;
BYTE pbKeyBlob[BLOB_SIZE];

BYTE pbCert[CERT_SIZE];
DWORD dwBlobLen;

BYTE pbHash[HASH_SIZE];
DWORD dwHashLen;
BYTE pbDestHash[HASH_SIZE];
DWORD dwDestHashLen;
HCRYPTHASH hHash = 0;
DWORD Certlength;
DWORD Certpartlength;

FILE *hSource = NULL;
FILE *hRemoteCert = NULL;
FILE *hDest = NULL;
FILE *hDataFile = NULL;
int eof = 0;

// room for padding, etc.
BYTE pbBuffer[BUFFER_SIZE];
char ReadBuffer[BUFFER_SIZE];
DWORD dwCount;

char localcertfilename[60], keyfilename[60], remotecertfilename[60];
char signaturekeyfilename[60];
char filetosend[60];
int scannedchar;
DWORD dwSourceLen;

```

```

int whicharg;

char serveripaddress[IP_ADDRESS_LEN];

// Read in IP address
// IP is defined with -i, DEFAULT otherwise
if (searchfor(argc, argv, "-i", &whicharg))
{
    strcpy(serveripaddress, (_strninc(argv[whicharg],2)));
}
else
{
    strcpy(serveripaddress, DEFAULT_SERVERIP);
}

if (!(askforserverip(serveripaddress)))
{
    goto done;
}

// Read in client (local) port number
// client port is defined with -p, DEFAULT otherwise
if (searchfor(argc, argv, "-p", &whicharg))
{
    localportnumber = atoi(_strninc(argv[whicharg],2));
}
else
{
    localportnumber = DEFAULT_CLIENT_PORT;
}

// Read in client (local) port number
// client port is defined with -p, DEFAULT otherwise
if (searchfor(argc, argv, "-s", &whicharg))
{
    connectportnumber = atoi(_strninc(argv[whicharg],2));
}
else
{
    connectportnumber = DEFAULT_SERVER_PORT;
}

// Read in LocalCertificate
// client port is defined with -m, DEFAULT otherwise
if (searchfor(argc, argv, "-m", &whicharg))
{
    if((hSource=fopen((_strninc(argv[whicharg],2)),"rb"))==NULL) {
printf("Error opening your certificate file!\n");
goto done;
    }
    strcpy(localcertfilename, (_strninc(argv[whicharg],2)));
}
else

```

```

    {
        if((hSource=fopen(DEFAULT_LOCAL_CERTFILE,"rb"))==NULL) {
printf("Error opening your certificate file!\n");
goto done;
        }
        strcpy(localcertfilename, DEFAULT_LOCAL_CERTFILE);
    }

// Read in junk filename
if (searchfor(argc, argv, "-j", &whicharg))
    {
        if((hRemoteCert=fopen((_strninc(argv[whicharg],2)), "wb"))==NULL) {
printf("Error opening Writing file!\n");
        }
        strcpy(remotecertfilename, (_strninc(argv[whicharg],2)));
    }
else
    {
        if((hRemoteCert=fopen(TEMPFILE,"wb"))==NULL) {
printf("Error opening certificate destination file!\n");
        }
        strcpy(remotecertfilename,TEMPFILE);
    }

// Get handle to the default provider.
if(!CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_FULL, 0)) {
    printf("Error %x during CryptAcquireContext!\n", GetLastError());
    goto done;
}

// Obtain the destination user's exchange public key. Import it into
// the CSP and place a handle to it in 'hDestPubKey'.

// Place the sending user's name in 'pbSendName' and set
// 'dwSendNameLen' to the number of bytes in it.

//Initiate connection

//Check versions of Winsock
wVersionRequested = MAKEWORD( 1, 1 );

err = WSAStartup( wVersionRequested, &wsaData );
if ( err != 0 ) {
    printf("Error %d during Startup\n", WSAGetLastError());

    /* Tell the user that we couldn't find a useable */
    /* WinSock DLL. */
    return;
}

/* Confirm that the WinSock DLL supports 2.0.*/
/* Note that if the DLL supports versions greater */
/* than 2.0 in addition to 2.0, it will still return */

```

```

/* 2.0 in wVersion since that is the version we      */
/* requested.                                     */

if ( LOBYTE( wsaData.wVersion ) != 1 ||
     HIBYTE( wsaData.wVersion ) != 1 ) {
    printf("Could not find useable Winsock DLL\n");
    /* Tell the user that we couldn't find a useable */
    /* WinSock DLL.                                 */
    WSACleanup( );
    return;
}

/* The WinSock DLL is acceptable. Proceed. */
connectedport = socket (AF_INET, SOCK_STREAM, 0);

if (connectedport < 0)
{
    printf("Error %d during create\n", WSAGetLastError());
    goto done;
}

printf("\nSocket created.\n");

serveraddr.sin_family    = AF_INET;
serveraddr.sin_addr.s_addr = inet_addr(serveripaddress);
serveraddr.sin_port      = htons ((short)connectportnumber);

clientaddr.sin_family    = AF_INET;
clientaddr.sin_addr.s_addr = htonl (INADDR_ANY);
//client.sin_addr.s_addr = inet_addr("18.172.0.225");
clientaddr.sin_port      = htons ((short)localportnumber);

if (bind(connectedport, (struct sockaddr *) &clientaddr, sizeof (clientaddr))<0)
{
    printf("Error %d during bind.\n", WSAGetLastError());
    goto done;
}

printf("Socket bound.\n");

if ((connecterror=connect(connectedport, (struct sockaddr *) &serveraddr, sizeof (serveraddr)))<0)
{
    printf("Error %d during connect.\n", WSAGetLastError());
    goto done;
}

printf("Connected to host.\n");

printf("Sending my certificate.\n");

// Get certificate length from file header
fread(&Certpartlength, sizeof(DWORD), 1, hSource);
Certlength=Certpartlength+sizeof(DWORD);

```

```

fread(&Certpartlength, sizeof(DWORD), 1, hSource);
Certlength=Certpartlength+Certlength+sizeof(DWORD);
fread(&Certpartlength, sizeof(DWORD), 1, hSource);
Certlength=Certpartlength+Certlength+sizeof(DWORD);
//rewind to start and read again
rewind(hSource);

dwCount = fread(pboutbuffer, 1, Certlength, hSource);
if(ferror(hSource)) {
    printf("Error reading data from source file!\n");
    goto done;
}
eof=feof(hSource);

if ((lettersent= sendto (connectedport, pboutbuffer, Certlength, 0,(struct sockaddr *) &clientaddr, sizeof
(clientaddr)))<0)
    //if ((lettersent= send (connectedport, outbuffer, strlen(outbuffer), 0))<0)
    {
        printf("Error %d during send.\n", WSAGetLastError());
        goto done;
    }

// Wait for the destination user to respond.

// Receive a key blob containing certificate

printf("Receiving remote certificate\n");

if ((lettersent= recv (connectedport, pbCert, CERT_SIZE, 0))<0)
    {
        printf("Error %d during receive of certificate.\n", WSAGetLastError());
        goto done;
    }

// Write data to destination file.
fwrite(pbCert, 1, lettersent, hRemoteCert);
if(ferror(hRemoteCert)) {
    printf("Error writing data to remote certificate file!\n");
    goto done;
}
if(hRemoteCert != NULL) fclose(hRemoteCert);

//Signature key filename -S
if (searchfor(argc, argv, "-S", &whicharg))
    {
        strcpy(signaturekeyfilename, (_strninc(argv[whicharg],2)));
    }
else
    {
        strcpy(signaturekeyfilename, DEFAULT_SIGNATURE_KEYFILE);
    }

```

```

if (!verifycertificate(hProv, remotecertfilename, signaturekeyfilename,
pbKeyBlob, &dwBlobLen,
pbSendName, &dwSendNameLen))
{
    printf("Could not verify certificate.\n");
    goto done;

}

// Import key blob into CSP.
if(!CryptImportKey(hProv, pbKeyBlob, dwBlobLen, 0, 0, &hDestPubKey)) {
    printf("Error %x during CryptImportKey!\n", GetLastError());
    goto done;
}

//Do I need this line? I'm overwriting the DestPubKey

if(!CryptGetUserKey(hProv, AT_KEYEXCHANGE, &hMyPubKey)) {
    printf("Error %x during GetUserKey!\n", GetLastError());
    goto done;
}

// Obtain the destination user's name. This is usually done at the
// same time as the public key was obtained. Place this in
// 'pbDestName' and set 'dwDestNameLen' to the number of bytes in
// the name.

//your name must be second item (item after function call)

if (argc<2)
{
    printf("No name specified.\n");
    goto done;
}
else
{
    strcpy(pbDestName, argv[1]);
    dwDestNameLen=strlen(pbDestName);
}

// Place the sending user's name in 'pbSendName' and set
// 'dwSendNameLen' to the number of bytes in it.

// Create a random session key (session key A). Because this key will
// be used solely for key exchange and not encryption, it
// does not matter which algorithm you specify here.

if(!CryptGenKey(hProv, CALG_RC2, CRYPT_EXPORTABLE, &hKeyA)) {
    printf("Error %x during CryptGenKey for KeyA!\n", GetLastError());
    goto done;
}

// Export session key A into a simple key blob.
dwBlobLen = BLOB_SIZE;

```

```

if(!CryptExportKey(hKeyA, hDestPubKey, SIMPLEBLOB, 0, pbKeyBlob, &dwBlobLen)) {
    printf("Error %x during CryptExportKey for KeyA!\n", GetLastError());
    goto done;
}

// Send key blob containing session key A to the destination user.

printf("Sending Session Key A.\n");

if ((lettersent= sendto (connectedport, pbKeyBlob, dwBlobLen, 0,(struct sockaddr *) &clientaddr, sizeof
(clientaddr)))<0)
    //if ((lettersent= send (connectedport, outbuffer, strlen(outbuffer), 0))<0)
    {
        printf("Error %d during send.\n", WSAGetLastError());
        goto done;
    }

// Wait for the destination user to respond.

// Receive a key blob containing session key B from the destination
// user and place it in 'pbKeyBlob'. Set 'dwBlobLen' to the number
// of bytes in the key blob.

printf("Receiving Session Key B\n\n");

if ((lettersent= recv (connectedport, pbKeyBlob, MAXLEN, 0)) < 0)
    {
        printf("Error %d during recv.\n", WSAGetLastError());
        goto done;
    }
dwBlobLen=lettersent;

// Receive a hash value from the destination user and place it in
// 'pbHashValue'. Set 'dwHashLen' to the number of bytes in the hash
// value.
printf("Receiving 1st hash value.\n");

if ((lettersent= recv (connectedport, pbDestHash, MAXLEN, 0)) < 0)
    {
        printf("Error %d during recv.\n", WSAGetLastError());
        goto done;
    }
dwDestHashLen=lettersent;

// Import the key blob into the CSP.

if(!CryptImportKey(hProv, pbKeyBlob, dwBlobLen, 0, 0, &hKeyB)) {
    printf("Error %x during CryptImportKey for KeyB!\n", GetLastError());
    goto done;
}

```

```

// Verify hash value received from the destination user.

// Create hash object.
if(!CryptCreateHash(hProv, CALG_MD5, 0, 0, &hHash)) {
    printf("Error %x during CryptCreateHash (to compare to received)\n", GetLastError());
    goto done;
}

// Add session key A to hash.
if(!CryptHashSessionKey(hHash, hKeyA, 0)) {
    printf("Error %x during CryptHashSessionKey (KeyA)\n", GetLastError());
    goto done;
}

// Add destination user's name to hash.
if(!CryptHashData(hHash, pbDestName, dwDestNameLen, 0)) {
    printf("Error %x during CryptHashData (destName)\n", GetLastError());
    goto done;
}

// Add session key B to hash.
if(!CryptHashSessionKey(hHash, hKeyB, 0)) {
    printf("Error %x during CryptHashSessionKey (KeyB)\n", GetLastError());
    goto done;
}

// Add sending user's name to hash.
if(!CryptHashData(hHash, pbSendName, dwSendNameLen, 0)) {
    printf("Error %x during CryptHashData (sendName)\n", GetLastError());
    goto done;
}

// Add "phase 2" text to hash.
if(!CryptHashData(hHash, "phase 2", 7, 0)) {
    printf("Error %x during CryptHashData (Phase 2)\n", GetLastError());
    goto done;
}

// Complete the hash computation and retrieve the hash value.
dwHashLen = HASH_SIZE;

if(!CryptGetHashParam(hHash, HP_HASHVAL, pbHash, &dwHashLen, 0)) {
    printf("Error %x during CryptGetHashParam!\n", GetLastError());
    goto done;
}

// Destroy the hash object.
if(!CryptDestroyHash(hHash)) {
    printf("Error %x during CryptDestroyHash!\n", GetLastError());
    goto done;
}

// Compare the hash value received from the destination user with

```

```

// the hash value that we just computed. If they do not match, then
// terminate the protocol.

if(dwHashLen!=dwDestHashLen || memcmp(pbHash, pbDestHash, dwHashLen)) {
    printf("Key exchange protocol failed in phase 2!\n");
    printf("Aborting protocol!\n");
    return;
}
printf("Hash values verified.\n");

// Compute hash to be sent to the destination user.
//
// Create hash object.

if(!CryptCreateHash(hProv, CALG_MD5, 0, 0, &hHash)) {
    printf("Error %x during CryptCreateHash (to send)!\n", GetLastError());
    goto done;
}

// Add session key B to hash.
if(!CryptHashSessionKey(hHash, hKeyB, 0)) {
    printf("Error %x during CryptHashSessionKey (KeyB)!\n", GetLastError());
    goto done;
}

// Add sending user's name to hash.
if(!CryptHashData(hHash, pbSendName, dwSendNameLen, 0)) {
    printf("Error %x during CryptHashData (SendName)!\n", GetLastError());
    goto done;
}

// Add destination user's name to hash.
if(!CryptHashData(hHash, pbDestName, dwDestNameLen, 0)) {
    printf("Error %x during CryptHashData (DestName)!\n", GetLastError());
    goto done;
}

// Add "phase 3" text to hash.
if(!CryptHashData(hHash, "phase 3", 7, 0)) {
    printf("Error %x during CryptHashSessionKey (phase3)!\n", GetLastError());
    goto done;
}

// Complete the hash computation and retrieve the hash value.
dwHashLen = HASH_SIZE;

if(!CryptGetHashParam(hHash, HP_HASHVAL, pbHash, &dwHashLen, 0)) {
    printf("Error %x during CryptGetHashParam!\n", GetLastError());
    goto done;
}

// Destroy the hash object.
if(!CryptDestroyHash(hHash)) {
    printf("Error %x during CryptGetHashParam!\n", GetLastError());
}

```

```

    goto done;
}

// Send the hash value to the destination user.

printf("Sending 2nd Hash Value\n\n");

if ((lettersent= sendto (connectedport, pbHash, dwHashLen, 0,(struct sockaddr *) &clientaddr, sizeof (cli-
entaddr)))<0)
    //if ((lettersent= send (connectedport, outbuffer, strlen(outbuffer), 0)<0)
    {
        printf("Error %d during send.\n", WSAGetLastError());
        goto done;
    }

// Create a shared session key to be used by both users for
// exchanging encrypted messages. Both users must agree on the
// algorithm and parameters that this key is to use.
//
// Create hash object.
if(!CryptCreateHash(hProv, CALG_MD5, 0, 0, &hHash)) {
    printf("Error %x during CryptCreateHash (for shared key)\n", GetLastError());
    goto done;
}

printf("Creating Shared Session Key\n");

// Add session key A to hash.
if(!CryptHashSessionKey(hHash, hKeyA, 0)) {
    printf("Error %x during CryptHashSessionKey (for shared key, hKeyA)\n", GetLastError());
    goto done;
}

// Add session key B to hash.
if(!CryptHashSessionKey(hHash, hKeyB, 0)) {
    printf("Error %x during CryptHashSessionKey (for shared key, hKeyB)\n", GetLastError());
    goto done;
}

// Complete the hash computation and derive a session key from it.
// The CRYPT_EXPORTABLE flag is not specified here because the key
// usually is not exported out of the CSP.

// changed to CRYPT_EXPORTABLE from 0

if(!CryptDeriveKey(hProv, CALG_RC4, hHash, CRYPT_EXPORTABLE, &hSharedKey)) {
    printf("Error %x during CryptDeriveKey!\n", GetLastError());
    goto done;
}

// Destroy the hash object.
if(!CryptDestroyHash(hHash)) {
    printf("Error %x during CryptDestroyHash!\n", GetLastError());
    goto done;
}

```

```

// Use the shared key to send encrypted messages to the other user.

printf("Successful Key Exchange.\n");

//get filename to encrypt & send

// Read in data file
if (argc<3)
{
    printf("No sending file specified!\n");
    goto done;
}

else
{
    if((hDataFile=fopen(argv[2],"rb"))==NULL) {
printf("Error opening file to send!\n");
    }

    //need to remove the path
    strcpy(filetosend,argv[2]);
}

// Encrypt source file and send it

dwSourceLen=0;
do {
    // Read up to BLOCK_SIZE bytes from source file.
    dwCount = fread(ReadBuffer, 1, BLOCK_SIZE, hDataFile);
    if(ferror(hDataFile)) {
        printf("Error reading data!\n");
        goto done;
    }
    eof=feof(hDataFile);

    dwSourceLen=dwSourceLen+dwCount;
    printf("source length %d\n",dwSourceLen);

} while(!feof(hDataFile));

// send file length
_itoa(dwSourceLen, pbBuffer, 10);
if ((lettersent= sendto (connectedport, pbBuffer, strlen(pbBuffer), 0,(struct sockaddr *) &clientaddr, sizeof
(clientaddr)))<0)
{
    printf("Error %d during send.\n", WSAGetLastError());
    goto done;
}

// other side is reading both buffers together
//install checkback
if ((lettersent= recv (connectedport, pbBuffer, 50, 0))<0)
{

```

```

    printf("Error %d during transmission of filelength data.\n", WSAGetLastError());
    goto done;
}

if (!(atol(pbBuffer)== (long)dwSourceLen)){
    printf("Data corrupted. Aborting.\n");
    goto done;
}

strcpy(pbBuffer,filetosend);
printf("filetosend: %s\n", filetosend);
// send file name
if ((lettersent= sendto (connectedport, pbBuffer, strlen(pbBuffer), 0,(struct sockaddr *) &clientaddr, sizeof
(clientaddr)))<0)
{
    printf("Error %d during send.\n", WSAGetLastError());
    goto done;
}

//verify correct name received
if ((lettersent= recv (connectedport, pbBuffer, 50, 0))<0)
{
    printf("Error %d during transmission of filename data.\n", WSAGetLastError());
    goto done;
}

if (!(strcmp(pbBuffer,filetosend)==0)){
    printf("Data corrupted. Aborting.\n");
    goto done;
}

printf("Sending file.\n");
rewind(hDataFile);
// send file
do {
    // Read up to BLOCK_SIZE bytes from source file.
    dwCount = fread(pbBuffer, 1, BLOCK_SIZE, hDataFile);
    if(ferror(hDataFile)) {
        printf("Error reading data!\n");
        goto done;
    }
    eof=feof(hDataFile);

    printf("pbBuffer: %s\n", pbBuffer);
    if(!CryptEncrypt(hSharedKey, 0, eof, 0, pbBuffer, &dwCount, BUFFER_SIZE)) {
        printf("Error %x during CryptEncrypt!\n", GetLastError());
        goto done;
    }

    if ((lettersent= sendto (connectedport, pbBuffer, (int)dwCount, 0,(struct sockaddr *) &clientaddr, sizeof
(clientaddr)))<0)
    {
        printf("Error %d during send.\n", WSAGetLastError());
        goto done;
    }
}

```

```

    }
} while(!feof(hDataFile));

done:

// Destroy session key.

if(!CryptDestroyKey(hSharedKey)) {
    printf("Error %x during CryptDestroyKey!\n", GetLastError());
    exit (2);
}

// Release provider handle.

if(!CryptReleaseContext(hProv, 0)) {
    printf("Error %x during CryptReleaseContext!\n", GetLastError());
    exit (2);
}

// Close source file.
if(hSource != NULL) fclose(hSource);

// Close Certificate file
if(hRemoteCert != NULL) fclose(hRemoteCert);

// Close data file
if(hDataFile != NULL) fclose(hDataFile);

// Close destination file.
if(hDest != NULL) fclose(hDest);

if (closesocket(connectedport) < 0)
{
    printf("Error %d during closesocket.\n", WSAGetLastError());
    exit (3);
}
}

```

## Bibliography

1. A. Agarwal, K. Hussein, A. Gupta, P. S. P. Wang, "Detection of Courtesy Amount Block on Bank Checks", *Journal of Electronic Imaging*. **5**, 2 (1996). 214-224.
2. K. Ashley. "The Pilot National Data Repository". March 1994.
3. C. Baru; R. Frost; J. Lopez; R. Marciano; R. Moore; A. Rajasekar; M. Wan, "Metadata Design for a Massive Data Analysis System", In Proc. CASCON '96 (11/96).
4. T. Boutelle. *World Wide Web Frequently Asked Questions*. 1994.
5. M. Ernst, "The Mechanization of Work", *Scientific American*. (1982). 133-165.
6. S. Glassman, M. Manasse, M. Abadi, P. Gauthier, P. Sobalvarro. *The Millicent Protocol for Inexpensive Electronic Commerce*. Proceedings of the 1st USENIX workshop on Electronic Commerce. July, 1995.
7. S. Genusa. *Using ISAPI*. 1997.
8. A. Gupta, M. V. Nagendraprasad, A. Liu, P. S. P. Wang and S. Ayyandurai, "An Integrated Architecture for Recognition of Totally Unconstrained Handwritten Numerals", *Int. J. Pattern Recogn. Artif. Intell.* **7**, 4 (1993). 113-129.
9. K. Hussein, A. Agarwal, A. Gupta, P. Wang, "A Knowledge Based Segmentation Algorithm for Enhanced Recognition of Handwritten Courtesy Amounts", Working Paper, MIT Sloan School of Management, 1993.
10. C. Leiserson, T. Cormen, R. Rivest. *Introduction to Algorithms*. The MIT Press. Cambridge, MA, 1990.
11. M. Manasse. *The Millicent protocols for electronic commerce*. The Proceedings of the 4th International World Wide Web Conference. December, 1995.
12. G. Medvinsky, B. Neuman. *NetCash: A design for practical electronic currency on the Internet*. Proceedings of 1st the ACM Conference on Computer and Communication Security, November 1993.

- 13.M. V. Nagendraprasad, P. L. Sparks, A. Gupta, "A Heuristic Multi-Stage Algorithm for Segmenting Simply Connected Handwritten Numerals", The Journal of Engineering and Technology. **6**, 4 (1993). 16-26.
- 14.M. V. Nagendraprasad, P. S. P. Wang, A. Gupta, "Algorithms for Thinning and Rethickening Binary Digital Patterns", Digital Signal Processing. **3** (1993). 97-102.
- 15.C. Neuman, G. Medvinsky. Requirements for Network Payment: The NetCheque Perspective. Proceedings of IEEE COMPCON'95, March 1995.
- 16.F. Pereira-Jesus. The Diffie-Hellman key exchange procedure. March 1995.
- 17.J. Saltzer. "A research prototype of the on-line electronic library of tomorrow". October 1991.
- 18.A. S. Tanenbaum. Modern Operating Systems. Prentice Hall. New Jersey, 1989.
- 19.Application Programmer's Guide Microsoft CryptoAPI. January 1996.
- 20.Lecture 10 of Computer and Network Security Class. MIT 1996.
- 21.National Institute of Standards and Technology (NIST). FIPS Publication 46-2: Data Encryption Standard. December 1993.
- 22.Object Oriented Programming class notes, <http://orange.mcs.dundee.ac.uk:8080/CS202/notes/mosaic/chap1.htm>
- 23.RFC 1867
- 24.RSA Data Security FAQ. 1997
- 25.Secure Electronic Transactions Specification. June 1996.
- 26.Winbank Project Highlights. 1995.