CITI Technical Report 00-6

Secure Internet Smartcards

Naomaru Itoi
Tomoko Fukuzawa
Peter Honeyman

Program in Smartcard Technology
`http://www.citi.umich.edu/projects/smartcard/`

Abstract

In this paper, we describe middleware that (1) enables secure communication between a host and a remote smartcard and (2) provides a unique name, regardless of card location.

Smartcards have traditionally been isolated from computer networks, communicating exclusively with the host computers to which they are attached through a serial port. This era is ending, in part due to the flexibility and programmability of JavaCards. Researchers are beginning to communicate with smartcards using Internet protocols.

This work extends the Internet infrastructure to allow secure access to remote smartcards, communicating encrypted payloads over UDP/IP. Session key establishment uses a PIN-based encrypted key exchange called SPEKE.

We describe two applications that use this infrastructure, Kerberos and SSH, discuss performance and security concerns, and highlight the security and convenience benefits of using Internet smartcards for personal key storage and cryptography.

August 24, 2000

Center for Information Technology Integration
University of Michigan
519 West William Street
Ann Arbor, MI 48103-4943

Secure Internet Smartcards

Naomaru Itoi
Tomoko Fukuzawa
Peter Honeyman

1 Introduction

Smartcards have long been used as tamper-resistant extensions to computers via exclusive access to a serial port. Smartcard applications, such as authentication systems [1, 12] and payment methods [6, 14], generally assume that smartcards and the applications that use them are on the same host. Current research is changing this situation by developing systems that use smartcards as remote servers [19, 20]. Extending the use of smartcards in this way removes the restriction that the smartcard always has to be on the same machine as the user. This can prove quite useful in some cases.

Consider a typical office, in which a user has several personal workstations providing diverse applications and services. If smartcards are used to enhance the security of such services as Kerberos authentication [22], `SSH` [24], or `PGP` digital signatures [26], a user is forced to install smartcard readers on all her computers and move cards around as her tasks demand.

We suggest a different scenario, in which smartcard functionality can be accessed from computers other than the one to which it is directly attached. Naturally, this poses some new security challenges, but also offers some interesting possibilities, such as keeping a smartcard itself physically secure in a locked room. In this case, a user is never able to use his smartcard without a way to access it remotely.

PC/SC-Lite [3] is among the more sophisticated card managers, in that it allows applications to access card readers on remote hosts. However, there are a few drawbacks to PC/SC-Lite. First, it does not encrypt data in transit between the smartcard and the remote host, and thus exposes potentially sensitive communication to Internet eavesdroppers. Second, the name of a smartcard depends on its location: a smartcard is identified by the host's domain name and a serial port on that host. If a card moves from one host to another or if the reader is moved to a different serial port, the smartcard's name becomes invalid.

In this paper, we describe a system that enables secure communication between a host and a remote smartcard and provides a unique name, regardless of card location. Our solution is based on adapting the smartcard web server developed at CITI [20] by adding UDP support and a protocol for secure, authenticated remote communication.

The building blocks of our system are listed below.

- A smartcard is given a (long-lived) domain name.
- A `UDP/IP` stack is implemented as a JavaCard applet.
- The *Simple Password Exponential Key Exchange* protocol (SPEKE) [13] is used to establish a session key between the remote user's host and the smartcard. Subsequent communication is encrypted with the session key.
- Kerberos and SSH clients are modified to communicate securely with an IP-based service that provides their key management and cryptographic needs.

Our middleware and sample applications demonstrate the convenience of the system and offer a development infrastructure for similar applications.

We assume that readers have a certain amount of knowledge about smartcards. Those who are not familiar with smartcard terminology are advised to consult the book by Guthery and Jurgensen [9].

2 Design

In this section, we describe the goals of our system and the decisions we made to achieve those goals.

2.1 Location independent naming

One of a smartcard's essential features is mobility. A smartcard owner can carry it around and use it at different locations. For maximum convenience, the name of the smartcard should not change when the smartcard moves; otherwise, the owner has to assign and remember multiple names. This can be a significant burden for the owner. Identifying a smartcard with DNS solves this problem by providing a smartcard with a location-independent name.

We assign a unique, durable, Internet domain name [17] to each card. The Internet domain name service (DNS) maps the domain name to an IP address. This assumes deployment of secure, dynamic DNS [23] or mobile IP [18].

By way of an example, the smartcard used in developing this project is always called `aya.citi.umich.edu`, no matter which workstation it is attached to.

2.2 Transport layer

Because an IP stack for JavaCard has been already developed in our lab [20], it is natural for us to choose UDP or TCP for data transmission. TCP has many advantages over UDP, namely: reliable, sequenced, error-free communication. Nonetheless, we elected to implement UDP because it is much simpler than TCP and has a smaller "footprint", essential for the limited hardware resources available on a smartcard. These limitations force CITI's TCP implementation to be less than complete, e.g., it does not retransmit dropped packets because smartcards lack an internal timer. The simplicity of UDP allows a more complete, standards-compliant implementation.

2.3 Security

Smartcard-based systems usually assume that the connection between a host and a smartcard reader is secure. This assumption is reasonable when the smartcard is attached to the local host over a serial line, which is hard to snoop or otherwise tamper with. The assumption no longer holds when part of the connection between a smartcard and a user's host is the Internet, which is generally an insecure medium. Consequently, our security goals require establishment of a secure channel between a host and a remote smartcard.

A secure channel has the following three properties: authenticity, secrecy, and integrity [4]. Our system achieves the first two properties by employing SPEKE, a secure key exchange protocol [13]. SPEKE establishes a session key for channel encryption while at the same time authenticating both parties with a shared secret.

We did not implement cryptographically secure integrity checking, or even UDP checksum in this implementation; at this time we find checksum calculation to be too time consuming for our applications. Our experience so far indicates that the lack of integrity checking does not have a detrimental impact on reliable communication. Implementing integrity checking would address one type of denial-of-service attack, but many others remain available to a powerful adversary in control of network traffic.

2.4 Alternatives to SPEKE

There are several alternatives to SPEKE, each with significant disadvantages.

- No encryption

  All messages are transmitted in the clear. This allows an adversary to eavesdrop and obtain all communication between the user and her smartcard.

- Sending a PIN

  The user sends a cleartext PIN to the Internet-attached smartcard and the card verifies it. This achieves authenticity, but allows an adversary to eavesdrop and steal the PIN.

- Encrypt with PIN

  A PIN can be used as a session key to encrypt the messages between the smartcard and the user's host. This achieves both authenticity and secrecy because it requires the parties to know the secret PIN. However, this is vulnerable to off-line guessing attacks: when a message contains identifiable strings, such as ASCII text or IP headers, an adversary can obtain the ciphertext and try all possible (exhaustive search) or likely (dictionary attack) PINs to decrypt the encrypted message. If a meaningful sequence of plaintext characters is uncovered, the PIN is revealed.

- Diffie-Hellman

  Diffie-Hellman key exchange (DH) can establish a session key between two parties [5]. However, it does not achieve authenticity, and is vulnerable to a man-in-the-middle attack [21].

- Encrypted Key Exchange

  Encrypted Key Exchange (EKE) [2] achieves both authenticity and secrecy and blunts the DH man-in-the-middle attacks by cleverly using a shared secret, even one that is susceptible to off-line attacks.

  EKE's patent holders did not not offer us permission to use the protocol.

- Open Key Exchange

  Open Key Exchange (OKE) [15] achieves the same goals as EKE and is not patented. Moreover, OKE is accompanied by a rigorous mathematical proof of its security properties. However, the protocol is fairly complicated and expensive, requiring modular multiplication, modular division, and three different hash functions; none of these is supported in the Schlumberger Cyberflex Access smartcard that we use.

2.5 SPEKE protocol

We settled on SPEKE, which achieves the same goals as EKE and OKE and can be implemented with resources available on smartcards. David Jablon generously permitted us to use SPEKE for non-commercial purposes.

SPEKE is a key exchange protocol based on Diffie-Hellman. SPEKE differs from DH mainly by using a shared secret to derive the base, instead of publishing the base in the initial exchange. This feature defeats the well-known man-in-the-middle attack on DH by forcing both parties to prove knowledge of a shared secret.

SPEKE computes the DH base by mapping the PIN to a base of prime order that is exponentiated by a random element. Without knowledge of the random exponent, an adversary is forced to compute the discrete log in order to gain information about the base; this is believed to be computationally prohibitive.

Because SPEKE does not use the shared secret to encrypt messages, it also avoids exposing plaintext/ciphertext pairs to off-line guessing attacks. SPEKE thus offers the essential properties we need to establish a secure channel.

The existence of a shared secret is reasonable for a system using smartcards: it is common practice to protect data in a smartcard with a personal identification number (PIN), which is a shared secret between the user and the smartcard. We return to this issue in Section 5.2.

Figure 1 summarizes the SPEKE protocol. The first stage of SPEKE uses the shared secret and DH to establish a session key. The session key may optionally be verified in the second stage by exchanging random challenges. Ker-

S	a secret shared between Alice and Bob
p	a prime number used as DH modulus
$f(S)$	a function that converts S into a suitable DH base
R_A, R_B	random numbers chosen by Alice and Bob
C_A, C_B	random challenges chosen by Alice and Bob
K	a session key generated as a result of SPEKE
$h(x)$	a one-way hash function, such as SHA1
$A \rightarrow B$: x	Alice sends x to Bob

DH Stage		
Step 1.	Alice computes $Q_A = f(S)^{R_A} \bmod p$	$A \rightarrow B$: Q_A
Step 2.	Bob computes $Q_B = f(S)^{R_B} \bmod p$	$B \rightarrow A$: Q_B
Step 3.	Alice computes $K = h(Q_B^{R_A} \bmod p)$	
Step 4.	Bob computes $K = h(Q_A^{R_B} \bmod p)$	
Verification (optional)		
Step 5.	Alice picks random number C_A	$A \rightarrow B$: $E_K(C_A)$
Step 6.	Bob picks random number C_B	$B \rightarrow A$: $E_K(C_A, C_B)$
Step 7.	Alice verifies C_A	$A \rightarrow B$: $E_K(C_B)$
Step 8.	Bob verifies C_B	

Figure 1: SPEKE protocol

beros and SSH are self-authenticating, so we omit consideration of this step in the remainder of this paper.

Figure 2 illustrates our design. $f(S)$ is precomputed and stored on the card. The host and the smartcard exchange two request/reply pairs, initiated by a connection request from the host. This signals the smartcard to generate its first message while a user is entering her PIN, possibly achieving some overlap.

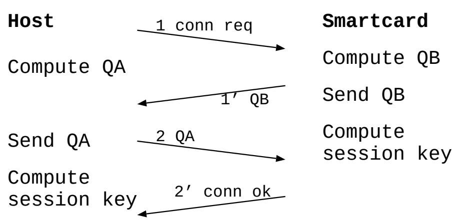

Figure 2: SPEKE implemented with smartcard

3 Implementation

3.1 Overview

Figure 3 illustrates the overview of our system. "Application" is a Kerberos or SSH client in our implementation. We modified off-the-shelf implementations of Kerberos and SSH to move their key management and cryptographic needs to a remote smartcard.

The Kerberos client uses the remote smartcard to unseal a DES-encrypted ticket granting ticket (TGT). The SSH client uses the remote smartcard to digitally sign a challenge presented by an SSH server. Viewed from a high level, these applications have similar needs, although they use vastly different base technologies.

The Kerberos or SSH client first establishes a session key with the remote smartcard using SPEKE, then exchanges messages with the smartcard to use its services. The messages are encrypted with the session key generated by SPEKE and transmitted by `UDP/IP`.

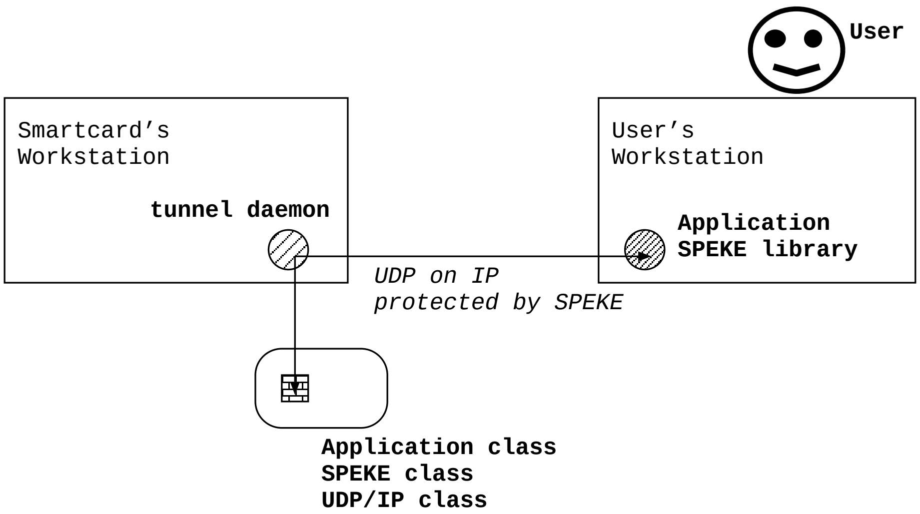

Figure 3: Implementation Overview

A daemon on the smartcard's host receives IP packets destined for the smartcard and forwards them to the smartcard through a "tunnel", which provides for proper framing of IP payloads in ISO 7816 Application Protocol Data Units (APDUs) [8].

Upon arrival of a message, the smartcard's `UDP/IP` class strips off the `IP` and `UDP` headers, and passes the datagram to the application class, which handles the request. The smartcard also has a `SPEKE` class, which plays the smartcard's part in session key establishment and message encryption.

We use `DES` for the encryption algorithm. The length of the modulus p is 1024-bit. The size of exponents R_A and R_B is 128-bit. Smaller exponents were tested, but did not significantly affect performance. Larger exponents would not improve security, given that we are using DES. The (public) modulus is a safe prime that is hard-coded on both ends. The base, derived from the shared secret, is precomputed on the card. Exponents and challenges are randomly generated in every session.

The host-side program was developed on Sun OS 5.6, and has been ported to Linux 2.2. The tunneling host runs OpenBSD 2.7 and has a Todos card reader attached. The card-side applet is written for the Schlumberger Cyberflex Access JavaCard. The Kerberos client is based on MIT distribution version 5-1.0.5, and SSH is based on SSH-1.2.27.

3.2 Component details

The system is divided into five components: host-side application, host-side SPEKE library, tunnel daemon, card-side application and card-side SPEKE library. This section details each component.

Host-side application

The host-side application is a program that provides service to a smartcard user. Our demonstration applications are Kerberos and SSH clients. In the remainder of this section, we describe the Kerberos implementation; SSH has the same basic issues.

The Kerberos client is a modified `kinit` program, which carries out user authentication with the Kerberos Key Distribution Center (KDC) using a user's key stored in a smartcard. This `kinit` is similar to the one developed at CITI

```
#define MAXDATASIZE 220  /* 248 - IP & UDP header length */
#define SPEKE_HDR_SIZE 3*sizeof(unsigned char)

typedef struct speke_t {
  unsigned char ver, msgid, len, data[MAXDATASIZE - SPEKE_HDR_SIZE];
} speke_t;
```

`ver` stores a constant (`0x10`) indicating the SPEKE protocol.

`msgid` identifies the type of a message. Possible values of `msgid` are `MSG_CONNECT`, `MSG_QB`, `MSG_QA`, `MSG_CHALLENGE_A`, `MSG_CHALLENGE_AB`, `MSG_CHALLENGE_B`, `MSG_REQUEST` and `MSG_REPLY`.

`len` is the length of data contained in the packet. Cyberflex DES methods require the data length to be a multiple of 8 bytes so the data may need to be padded. `len` indicates the logical length of the data.

`data` is the data to be transmitted

Figure 4: SPEKE data structure

for Kerberos/smartcard integration using a local smartcard [12], except that it uses UDP/IP to communicate with a smartcard on a remote host instead of communicating conventional APDUs over a serial port to a locally-attached reader. Where the earlier smartcard-enabled `kinit` uses CITI's sc7816 communication library [20], the implementation described here uses our SPEKE library.

The Kerberos client follows three steps to receive the service from the smartcard: (1) establish a session key, (2) get a principal name × key number table from the smartcard, and (3) use the smartcard to decrypt the reply from the KDC. The first step is accomplished by calling `speke_connect()`. The others use `speke_send()` and `speke_recv()`.

Here is an example of calls to the SPEKE library. `speke` is a data structure that stores the context of a SPEKE session. `sockfd` is a socket descriptor used to communicate with a smartcard.

```
/* key establishment */
speke_open (&speke, sockfd, hostname,
  SERV_PORT);

/* send 2 bytes to card */
n = speke_send (&speke, sockfd, bufr, 2);

/* receive up to 256 bytes from card */
n = speke_recv (&speke, sockfd, bufr, 256);
```

SPEKE library

This host-side library implements the SPEKE key exchange protocol and exports procedures for connection establishment, connection destruction, and data transmission. The roles of these functions are summarized below.

`speke_open` asks the user for a PIN and establishes a session key using SPEKE

`speke_send` encrypts and sends data to the card

`speke_recv` receives and decrypts data from the card

`speke_close` destroys the session key

The SPEKE library uses UDP/IP for data transport. All the SPEKE packets sent and received by the SPEKE library are UDP datagrams, with the format depicted in Figure 4.

SPEKE uses several cryptographic operations, such as DES, modular exponentiation, and SHA1. The host-side SPEKE library includes three libraries to enable these operations: a DES library (libdes-4.01) by Eric Young [25], the GNU Multiple Precision Arithmetic Library [7], and the CTC library [16].

Tunnel daemon

The tunnel daemon is the only component that runs on the smartcard's host computer. The job of the tunnel daemon is to encapsulate IP packets into APDUs, which smartcards understand. The routing table of the smartcard's host is configured so that the tunnel daemon receives packets directed to the smartcard's IP address.

When the tunnel daemon receives an IP packet, it prepends a 5-byte APDU header to it, and sends the APDU to the smartcard using CITI's sc7816 library. If a reply packet is available from the smartcard, the tunnel daemon issues a `get-response` APDU to the smartcard.

After receiving a response packet from the card, the tunnel daemon strips the APDU header and transmits the payload to the address specified in the IP header. Beyond this, tunnel daemon operation does not depend on the IP payload; it merely attaches and strips APDU headers and routes IP packets between the network and a serial device.

Card-side application

The remaining three components run on the smartcard. The highest level component among them is a card-side application program that provides application-specific services. For example, the Kerberos application decrypts a message that is encrypted with a user's symmetric key; the SSH application encrypts a random challenge with a private RSA key.

Thanks to the object oriented style of programming supported by JavaCards and to the SPEKE class taking care of the details of key exchange and message secrecy, all an application class has to do for communication is to inherit the SPEKE class and issue `send()` and `recv()` methods. An example follows:

```
public class KrbSpeke extends UdpSpeke {
  public void process(APDU apdu) {
    short len = recv(apdu);
    if (len >= 0) {
      len = kerberos_process (apdu, len);
      send (apdu, len);
    }
  }
}
```

SPEKE class

The next layer on the smartcard is the SPEKE class. Similar to the host-side SPEKE library, the SPEKE class implements the SPEKE key exchange protocol and exports methods for data transmission. The API consists of two methods, `send()` and `recv()`:

`recv` parses a packet. If the packet contains a message for key exchange, this method creates an appropriate packet and sends it out. Otherwise (i.e., if the packet carries data), it decrypts the data and passes it to the application class.

`send` encrypts a message and sends it.

The SPEKE class inherits the UDP class.

```
public class UdpSpeke extends Udp7816 {
  void send (APDU apdu, short len);
  short recv (APDU apdu);
}
```

UDP/IP class

The last component is the UDP/IP class, which processes UDP/IP datagrams. This is built on

CITI's smartcard IP stack. For incoming packets, the `recv()` method strips off UDP and IP headers and hands the data to the upper layer, in our case, the SPEKE class. Packets transmitted in the other direction are handled by the `send()` method, which adds UDP and IP headers to a message and sends it out of the smartcard.

4 Performance

In this section, we discuss performance. This system is not fast. Therefore, we focus on highlighting system bottlenecks, and discuss how performance can be improved.

Performance evaluation was carried out on two workstations in our LAN. The user's workstation is Linux 2.2 on a 400 MHz Pentium, and the smartcard's workstation is OpenBSD 2.7 on a 400 MHz Pentium. The smartcard is attached to the workstation with a Litronic PC-3 reader communicating at 115 Kbps.

4.1 Execution time

The table shows the execution time of Kerberos and SSH client programs using our SPEKE library. The performance results of clients that use local smartcards and the sc7816 library are shown for comparison. All times are reported in seconds and are the average of five time trials. Variance is negligible.

	remote	local
Kerberos	12.8	3.33
SSH	12.6	3.43

The remote versions are much slower than the local ones. The difference is due largely to two factors: setting up SPEKE and the cost of encrypting and decrypting payloads. The next section focuses on these two factors.

4.2 Details

In this section, we discuss execution of the Kerberos client. Our observations also apply to the SSH client.

The events in the Kerberos client are listed below in chronological order. All times are reported in seconds and are the average of five time trials. Variance is negligible.

time	events
0.00	kinit start
0.02	SPEKE connect start
0.03	Host send SPEKE1 (connect request)
0.03	Host send SPEKE2 (Q_A)
2.07	Host recv SPEKE1 (Q_B)
3.56	Host recv SPEKE2 (connect ok)
3.56	get_key_num start
5.88	get_key_num finish
5.88	decrypt ticket start
9.93	decrypt ticket finish
9.93	decrypt ticket start
12.80	decrypt ticket finish
12.80	kinit end

Data to be decrypted is divided into two blocks and sent separately because, at 224 bytes, a Kerberos ticket is too large for a smartcard to decrypt at once.

Within the total 12.80 seconds, time for using smartcard dominates, taking 12.78 seconds. This is not surprising: it takes 2 – 4 seconds to exchange a pair of request-reply packets, and there are five such pairs:

request type	time
SPEKE1 ($\rightarrow$ connect request, $\leftarrow Q_B$)	2.04
SPEKE2 ($\rightarrow Q_A$, $\leftarrow$ connect ok)	1.49
get_key_num request ($\leftarrow$ princ table)	2.33
decrypt block 1 ($\rightarrow$ data, $\leftarrow$ data)	4.06
decrypt block 2 ($\rightarrow$ data, $\leftarrow$ data)	2.87

Now we analyze the message exchange bottleneck. Processing a request is divided into five phases.

- time spent in the smartcard
- IP communication between user's host and

S	a secret shared between Alice and Bob
P	a public key generated by Alice
K	a session key generated as a result of EKE
$A \to B$: x	Alice sends x to Bob

Step 1.	Alice generates P, encrypts it with S	$A \to B$: $E_S(P)$
Step 2.	Bob obtains P, generates K, computes $E_S(E_P(K))$	$B \to A$: $E_S(E_P(K))$

Figure 5: EKE protocol key exchange. The optional verification step is not shown.

smartcard's host

- overhead of the tunnel daemon
- sc7816 library overhead
- communication between smartcard and smartcard's host. (This includes time for `get-response` APDU.)

Using the first message, SPEKE1, as a sample, we measure the following events. All times are reported in seconds and are the average of five time trials. Variance is negligible.

event	time
IP + tunnel + sc7816	0.00
in-the-card	1.83
card communication	0.21
Total	2.04

Execution time in the smartcard dominates the other parts with a ratio of 9:1. Cryptographic operations, such as modular exponentiation by an RSA method, DES, and random number generation, are the main reasons that it takes so much time in the smartcard. Significant improvement in performance of our system is impossible without a faster smartcard or a protocol that is less computationally demanding.

Card communication time can be reduced with the `T=1` protocol instead of `T=0`. With `T=0`, a `get-response` APDU is necessary to obtain data returned from a smartcard in addition to a service request APDU. With `T=1`, the smartcard returns data immediately after a request is made, eliminating the overhead of the `get-response` APDU. The Cyberflex Access smartcards we use do not support `T=1`.

4.3 EKE measurement

Although we cannot use it in our projects because of a patent issue, we implemented and measured EKE to satisfy our curiosity. EKE is a simple and well-known protocol. The EKE protocol, described in Figure 5, is implemented with one pair of messages and optional verification. Like our SPEKE implementation, we initiate EKE with a connection request, which allows the smartcard to overlap its random number generation with the host's key pair generation and PIN input. The first message, `EKE1`, requests connection. The smartcard starts generating random numbers after receiving it. The second message, `EKE2`, implements steps 1 and 2.

A chronological event list is shown below. EKE takes 4.47 seconds to complete connection establishment, compared to 3.56 seconds for SPEKE.

time	events
0.00	EKE connect start
0.01	Host send EKE1 (connect request)
1.43	Host send EKE2 ($E_S(P)$)
4.45	Host recv EKE2 ($E_S(E_P(K))$
4.47	EKE connect ok

Time taken for each message pair is as follows.

request type	time
EKE1 ($\to$ connect request, $\leftarrow$ `NULL`)	0.83
EKE2 ($\to E_S(P)$, $\leftarrow E_S(E_P(K))$)	2.95

Although EKE is simpler than SPEKE, the time required to generate a key pair on the host (approximately 1.5 sec) hurts performance. Moore's law influences key generation time, but this is moderated by the fact that faster computers demand longer keys, which take longer to generate. On the whole, though, we expect key generation time to improve with new generations of microprocessors.

5 Discussion

In this section, we summarize the advantages of using Internet-attached smartcards and discuss security considerations.

5.1 Summary

The following four aspects highlight the value of this work.

Useful and necessary. Last year, we implemented and deployed our sc7816 version of the smartcard-integrated Kerberos to staff at CITI. CITI staff frequently use a lot of different workstations. It quickly became clear that accessing a smartcard remotely would extend the benefit of smartcard-enabled Kerberos to all our computers while saving us from having to install a reader on each of them.

The first application of smartcard IP for personal usage. We find smartcards very effective when used as personal security devices connected to the Internet. Our work is the first implementation of such a system (that we know of).

Standard API. Our protocols are built on UDP and IP, universally accepted communication standards. We hope to positively influence today's smartcard API woe: many smartcard APIs are proposed, but none has established dominance, forcing developers to learn API after API.

Development framework. Our system enables developers to implement IP-based smartcard applications easily. Our source code is freely available on the CITI smartcard web page.

5.2 Security considerations

In this section, we consider two potential vulnerabilities that could compromise the security of Internet smartcard: host compromise and online attack. Regarding the former, entering a PIN or pass phrase on a workstation is always fraught with vulnerabilities should the workstation be compromised. Most workstations do not provide a "trusted path" from the keyboard to a secure application.

As for the latter, Jablon suggests a number of methods that can be used to prevent information leakage in SPEKE (and EKE) [13]; although we have not implemented all of these, we would find it prudent to do so before fully deploying secure Internet smartcards.

Following Jablon's suggestions, we feel confident that SPEKE can effectively blunt off-line guessing attacks on the PIN. However, on-line attack remains a potent threat. An on-line attack would proceed as follows. Each candidate PIN can be used by an adversary to establish a secure channel. Subsequent use of the channel either reveals a cleartext Kerberos TGT (or properly signed SSH nonce) or random garbage. Eventually, the adversary will try the correct PIN and defeat the PIN-based security of the system.

In our system, we use four-digit PINs. If we assume that a session can be completed in five seconds, then the entire space of potential PINs can be tested by on on-line adversary in 50,000 seconds, less than a day if the card is kept online.

To blunt this sort of attack, we suggest a counter on the card that keeps track of failed attempts to complete a Kerberos or SSH session. However, this presents an obvious denial-of-service attack, so we are considering adding an administrative interface that uses a strong key to allow the counter to be reset remotely.

Alternatively, the PIN space could be expanded; a seven-digit PIN would require over a year of continuous testing to search the entire space. The S/KEY one-time password system [10, 11] represents random keys by selecting short phrases made up of taken from a 2,048 word dictionary, e.g., "WAIT POD LIMA." Each word contributes 11 bits to the size of the search space; a three-word phrase would require centuries to search the entire space.

6 Conclusion

We designed and implemented an Internet-standards compliant middleware infrastructure that provides secure access to remote smartcards, and built two demonstration applications on it. The performance of the system reflects the performance realities of today's smartcards. Yet, we find the infrastructure useful, and anticipate that it will enable many new types of smartcard applications.

7 Acknowledgment

This work extends Jim Rees' pioneering implementation of Internet protocols on a smartcard. We thank Dug Song, Niels Provos, Wolfgang Ley, Gasper Carson, and Angelos Keromytis for valuable discussions. David Jablon kindly allowed us to use the SPEKE protocol.

This work was partially supported by a research grant from Schlumberger, Inc.

References

[1] Bastiaan Bakker. Mutual authentication with smart cards. In *USENIX Workshop on Smartcard Technology*, Chicago, May 1999.

[2] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84, Oakland, May 1992.

[3] David Corcoran. MUSCLE: Movement for the use of smart cards in a Linux environment. `www.linuxnet.com`.

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

[5] W. Diffie and M. E. Hellman. New directions in cryptography. In *IEEE Trans. Inform. Theory*, volume IT-22(6), pages 644–654, Nov 1976.

[6] Europay, MasterCard, and Visa. EMV'96: Integrated circuit card application specification for payment systems, June 1996. `www.mastercard.com/emv`.

[7] GNU multiple precision arithmetic library. `www.swox.com/gmp`.

[8] S. Guthery, Y. Baudoin, J. Posegga, and J. Rees. IP and ARP over ISO 7816-3 (Internet Draft), February 2000.

[9] Scott B. Guthery and Timothy M. Jurgensen. *Smart Card Developer's Kit*. MacMillan Technical Publishing, Indianapolis, December 1997.

[10] N. Haller. The s/key one-time password system, RFC 1760, Feb. 1995.

[11] N. Haller and C. Metz. A one-time password system, RFC 1938, May 1996.

[12] Naomaru Itoi and Peter Honeyman. Smartcard integration with Kerberos V5. In *USENIX Workshop on Smartcard Technology*, Chicago, May 1999.

[13] David P. Jablon. Strong password-only authenticated key exchange. *ACM Computer Communications Review*, October 1996.

[14] Secure Electronic Transaction LLC. SET standard technical specifications, 1999. `www.setco.org` .

[15] Stephan Lucks. Open Key Exchange: How to defeat dictionary attacks without encrypting public keys. In *Security Protocol Workshop '97*, Ecole Normale Superieure, April 1997.

[16] Ian Miller and Mr. Tines. CTC library. `www.bifroest.demon.co.uk/ctc/`.

[17] Paul Mockapetris. Domain names - concepts and facilities, STD 13, RFC 1034, Nov. 1987.

[18] C. Perkins. IP mobility support. RFC 2002, October 1996.

[19] Joachim Posegga and Scott Guthery. How to turn a GSM SIM into a web server. In *CARDIS 2000*, Bristol, September 2000.

[20] Jim Rees and Peter Honeyman. Webcard: A Java Card web server. In *CARDIS 2000*, Bristol, September 2000.

[21] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, 2nd edition, 1996.

[22] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. In *Winter 1988 USENIX Conference*, Dallas, February 1988.

[23] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. Dynamic updates in the domain name system (DNS update). RFC 2136, April 1997.

[24] Tatu Ylönen. Ssh – secure login connections over the internet. In *6th USENIX Unix Security Symposium*, July 1996.

[25] Eric Young. libdes DES library. ftp://ftp.psy.uq.oz.au/pub/Crypto/DES.

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