1 Introduction

IPSec is an open draft standard for packet level security in IP networks. It intends to provide protection against eavesdropping and data tampering by third parties, features the current IP infrastructure is desparately lacking. IPSec is a standard part of the IPv6 protocol, therefore its presence in the infrastructure of the internet will be pervasive once IPv6 fully is enrolled. But IPSec is not restricted to IPv6, currently most implementations run on IPv4.

IPSec is under active development by the IETF IPSec working group. At least partly because the IPSec standards are the result of a committee the standards tend to be overly complex and contain way to many features and options. Ferguson and Schneier evaluated the IPSec specifications in [FeSch00]. Their biggest complaint against IPSec was the aforementioned complexity. Ironically in this paper they both recommend against using IPSec and in favor of using IPSec. The dilemma is caused by the fact that all alternatives to IPSec are far worse (security wise).

FreeS/WAN is an open source implementation of IPSec for the Linux operating system. John Gilmore of the Electronic Frontier Foundation started the project to answer his self set goal to protect internet traffic against passive wiretapping. The code is under active development by a team of volunteers from all over the world. Unlike other IPSec implementations FreeS/WAN does not suffer from the cryptography export regulations of the US: all code is maintained in a free country outside the US and the development team pays significant attention to staying 'export clean'[1].

The basic components of FreeS/WAN are depicted in figure 1, which shows two hosts, West Gate and East Gate, that communicate with each other via IPSec. Typically at least one of the machines is a gateway to an internal subnet. Together the gates and the subnet create one Virtual Private Network (VPN) that is securely interconnected over the internet.

For IPSec the IP connection carries two types of traffic:

- ESP, which stands for Encrypted Security Payload. This is the encrypted IP traffic between the hosts

- IKE, which stands for Internet Key Exchange. This the protocol by which session keys of ESP are exchanged.

[1]Amongst others this means accepting no source code from Americans. The US have relaxed their export regulations January 2000 and now allow open source projects to export strong cryptography. Because of that the team may relax its policy in the near future.

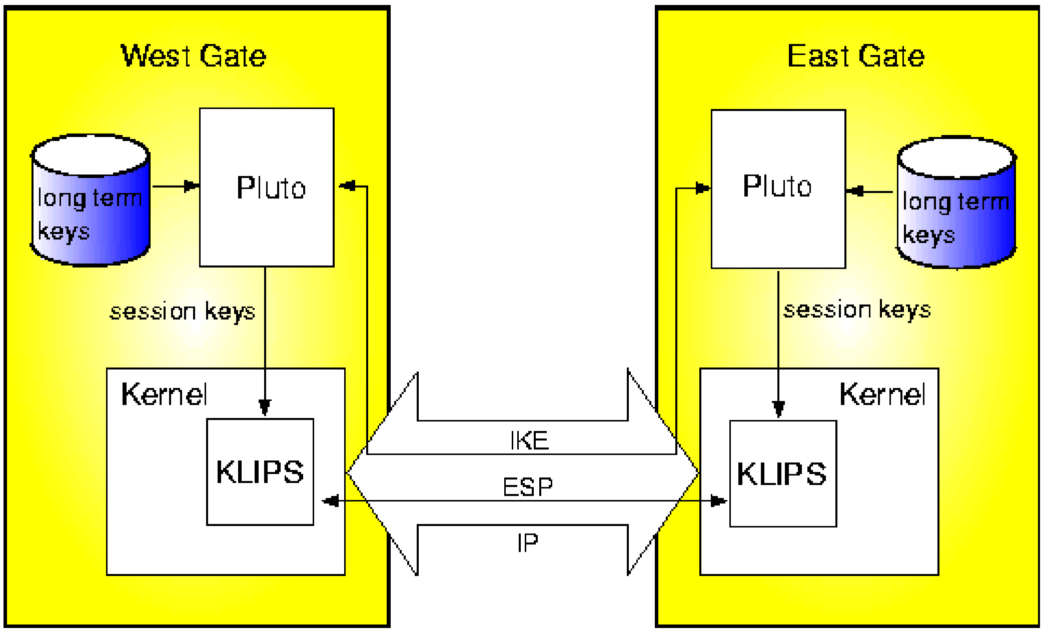

Figure 1: FreeS/WAN components

The KLIPS module inside the Linux kernel performs the encryption of the IP data to ESP and the decryption of ESP to IP data. The session key exchanges are handled by a user space process called Pluto. Pluto has a list of long term keys to authenticate these key exchanges. Currently these long term keys are stored on harddisk. The goal of this paper is to investigate other safer means to store these secrets.

2 Key management in IPSec

FreeS/WAN has two keying methods: manual keying and automatic keying. With manual keying the session key itself is specified in the configuration file (/etc/ipsec.conf). Upon initialization FreeS/WAN passes this key directly to the kernel level IP encryption module. FreeS/WAN performs no key management at all, that burden is left to the administrator. This keying method is only intended for debugging purposes: since there is no key management it is convenient for checking the setup without worrying whether both gates use the same session key. However, it is not very secure: anyone who can obtain the key can decrypt all past and future communication simply by eavesdropping.

The second method, automatic keying, is more sophisticated: using long term keys, a daemon process, the Pluto daemon, regularly negotiates session keys with its peer. Traffic will never be encrypted with the long term key only with the session keys. IPSec peers negotiate their session keys with the Internet Key Exchange protocol [IKE].

2.1 the Internet Key Exchange

IKE is an elaborate protocol for key exchanges, with multiple multiple modes and phases. IKE defines key negotiation based on pre-shared secrets, digital signatures and public key encryption. FreeS/WANs IKE implementation, the Pluto daemon, has supported the pre-shared secret method since its first release. Recently support for key negotiation based on public key encryption has been added.

IKE uses the Diffie-Hellman key exchange [DH] to establish a shared secret between the two peers. With Diffie-Hellman no pre-shared key is necessary to negotiate this secret. But what good is a negotiated secret if you don't know with whom you share it? This is where the pre-shared secret or public key crypto key pairs come into play: with these the IKE daemons each calculate an authentication code, called the initiator digest and responder digest respectively.

The exact formula's for shared secrets based session key calculation are given in algorithm 1. Several keys are defined:

- $SKEYID$ is the primary session key, from which all other session keys are derived.
- $SKEYID_D$ is keying material used to derive keys for security services during phase 2 of IKE.
- $SKEYID_A$ is the keying material used by the parties of the exchange for message integrity.
- $SKEYID_E$ is the keying material used by the parties of the exchange to protect the confidentiality of messages.

The 'prf' function in these calculations denotes a pseudo-random function'. IKE actually uses the HMAC-SHA-1 construction by default, which provides the security we need, if the hash function SHA-1 is secure.

The calculations use the following parameters:

- N_i and N_r are two nonces generated by the initiator and responder respectively.

Algorithm 1 calculation of session keys and digests in IKE using shared secrets.

```
$SKEYID = prf(pre-shared-key, N_i|N_r)$
$SKEYID_D = prf(SKEYID, g^{ir}|CKY_I|CKY_R|0)$
$SKEYID_A = prf(SKEYID, SKEYID_D, g^{ir}|CKY_I|CKY_R|1)$
$SKEYID_E = prf(SKEYID, SKEYID_D, g^{ir}|CKY_I|CKY_R|2)$
$Idigest = prf(SKEYID, g^i|g^r|CKY_I|CKY_R|SA_{ib}|ID_{i1b})$
$Rdigest = prf(SKEYID, g^r|g^i|CKY_R|CKY_I|SA_{ib}|ID_{r1b})$
```

- g^i, g^r and g^{ir} are respectively the Diffie-Hellman public values of the initiator and responder and the exchanged Diffie-Hellman secret.
- CKY_I and CKY_R are the cookies generated by the initiator and responder that identify the Secure Association.
- SA_{ib} is the entire body of the 'SA payload'. This payload contains the Domain of Interpretation (DOI) and the encryption proposals amongst others.

IKE exchanges these parameters in the following protocol flow:

Initiator		Responder
SA	$\longrightarrow$	
	$\longleftarrow$	SA
g^i, N_i	$\longrightarrow$	
	$\longleftarrow$	g^r, N_r
$E(ID_{i1}, Idigest)$	$\longrightarrow$	
	$\longleftarrow$	$E(ID_{r1}, Rdigest)$

The last two messages are encrypted with a key derived from $SKEYID_E$, thus shielding the IDs of both parties from eavesdropping.

An important property of the IKE protocol is Perfect Forward Secrecy (PFS). This is the notion that compromise of a single key will permit access only to data protected by a single key[IKE]. In effect this means that compromise of the long term key (the pre-shared secret) will not make the system vulnerable to passive attacks: from this key and eavesdropped communication alone the attacker cannot deduce the data encryption keys of a session. Knowledge of the long term keys allows an active man in the middle attack however. Of course with knowledge of the long term keys an attacker also can pretend to be one of the legitimate parties and initiate a secure connection.

Certainly the protection of the automatic keying mechanism is much better than the security of the manual keying method, but nevertheless compromise of the long term keys has serious security implications.

3 Methods for improving the security of shared secrets

To improve the security of FreeS/WAN against active attacks we need to better protect our long term keys. Sure, Linux' standard file system, ext2, has access controls that can prevent anyone but the root user to read the IPSec secrets file (/etc/ipsec.secrets). But that will not help very much once someone has hacked the root account or the machine has been stolen. Or a disgruntled (ex)employee with root privileged may have made a copy of the file.

3.1 Hardware tokens

Smartcards are a popular form of hardware tokens. In the Netherlands alone over 26 million smartcards[2] are in use, on a population of 16 million people. Smartcards basically are fully integrated computers (CPU, ROM, RAM and I/O) in a single chip glued on a credit card format plastic card. ISO has defined standards for smartcards in ISO7816, ranging from physical and electrical characteristics in part 1 and 2, data communication in part 3, application command in part 4, to services in part 5, 6 and 7. Almost all cards implement (at least part of) the command set described in ISO7816-4, parts 5 and higher often are not implemented. They offer a simple hierarchical file system with access protection (with passwords or challenge/response) and content authentication (through keyed Message Authentication Codes).

Mostly because of political reasons, smartcards traditionally have been very restrictive: it was not possible to upload your own programs on your card, even writing your own keys on the card often was limited.

Most card manufacturers now also offer Java Card compliant smartcards. Java Card technology, introduced by SUN Microsystems, offers much more

[2] 13.400.000 Chipknip cards [Ipay] + 7.000.000 chipper cards [Chip] + 6.000.000 GSM SIM cards.

freedom: developers can write little programs (applets) that can be loaded in the card and executed. Some other benefits Sun mentions are:

Platform Independence

Java Card technology applets that comply with the Java Card API specification will run on cards developed using Java Card Application Environment. This way developers can reuse the same applet with cards from different vendors.

Multi-Application Capable

Multiple applications can run on a single card. The Java runtime environment in the card ensures that multiple applications can securely reside and execute on the same card.

Post-Issuance of Applications

Applications can be installed after the card has been issued. This enables card issuers to dynamically respond to their customer's changing needs.

Flexibility

The object oriented methodology of the Java Card technology provides flexibility in programming smartcards.

Compatibility with Existing Smartcard Standards

The Java Card API is compatible with formal international standards such as ISO7816 and industry-specific standards (e.g. EMV).

3.2 The Dallas Semiconductor iButton

The iButton is a standard JAVA "card" in a 16 mm, stainless steel case. Some accessories enable one to to wear the iButton, like metal cards, watches, or finger rings. Note that the "wearability" is not a a funky feature as user friendly constructions are absolutely vital for secure systems.

"While cards are fine for playing poker, they're not a safe place to keep a fragile chip that defines your digital identity." [Dallas]

The unusual form of the i-Button provides reasonable security against hardware attacks, compared to the security of ordinary chip cards [WTK97]. Furthermore the case provides clear visual evidence of tampering.

Dallas Semiconductor [Dallas] gives the following summary of the physical security:

- Armored with stainless steel for the hard knocks of everyday use
- Wear tested for 1 million insertions and more than 10 years of life
- ESD protection is more than 25,000 volts for wash-and-wear dependability
- Three-layer metal technology and flip-chip bonding form barricades to protect data
- Opening of the physical perimeter generates a tamper response
- Tamper response causes rapid zeroization of NV SRAM to prevent disclosure of secure data.

The 6 KB of SRAM included on the monolithic chip has been specially designed so that it will rapidly erase its contents in the event of an intrusion. The following instances are treated as intrusions:

- Opening the case
- Removing the metallurgically bonded substrate barricade
- Micro-probing the chip
- Subjecting the chip to temperature extremes.

There are several design elements against Differential Fault Analysis. Thus if excessive voltage is encountered, the sole I/O pin is designed to fuse. This will render the chip inoperable.

The National Institute of Standards (NIST) and the Canadian Security Establishment (CSE) have validated a version of the crypto i-Button for protection of sensitive, unclassified information. FIPS 140-1 validation assures government agencies that the products provide a trusted, physically secure module to properly protect secure information.

According to Dallas Semiconductor, over 27 million i-Buttons are currently (Nov. 1999) in circulation.

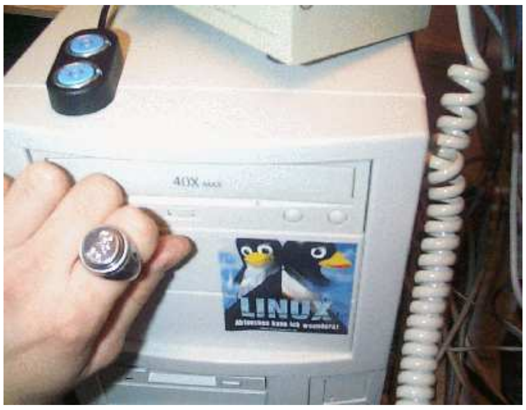

Figure 2: the Dallas Semiconductor iButton and Blue Dot Receptor

3.3 The smartcard file system method

The simplest proposed method to let the secrets reside in a hardware token is to use the token as a mobile private file system. The Pluto daemon will simply read out the long term key from the button when it needs it instead of from /etc/ipsec.secrets. This could be implemented even without modification of the Pluto implementation: a new 'smartcard' file system in the Linux kernel could mount the data files from the smartcard into the normal virtual file system hierarchy. Pluto then can simply read the keys like a normal file, without even knowing the keys are not stored on hard disk. Itoi, e.a. explored the idea for a 'UNIX smartcard file system in [IHR98]. They built an implementation for OpenBSD and which is now successfully used by the authors to store their private Secure Shell (SSH) keys on smartcards.

This method clearly is an improvement over storing keys on the hard disk, but when card is inserted the key still is vulnerable.

3.4 SKEYID calculation on the card method

A better security model has the following minimum requirements for the protocol[Weis00]:

- The secret master key *must never* leave the card.
- The protocol has to be secure against a master key recovery attack.
- It's not feasible to decrypt traffic without breaking the host cipher or the pseudorandom function on the card.

Furthermore it is highly preferable that the protocol not only fits within the IPSec framework but is compatible with it on the wire level: the protocol should allow interoperability with other IPSec implementations without their modification.

The IPSec working group has not made recommendations on secure storage of IPSec's secrets. However, the session key calculations IKE defines provide a logical approach: IKE uses the long term key at only one point, in the calculation of the SKEYID. It makes sense to perform this in the card and all further session key calculations in the host. Calculating $SKEYID_D$, $SKEYID_A$, etc. in the card does not improve the security of the shared secret and requires extra CPU resources of the relatively slow smartcard.

The security of this scheme relies on the security of the HMAC calculation. It shall not be possible to recover the secret key in a chosen plaintext attack, ie. against an adversary that can send arbitrary nonces to the card and read back the results. The security of HMAC relies on relatively weak assumptions on the underlying hash function. IKE uses SHA1 as underlying hash, which satisfies these assumptions. The security of HMAC-SHA1 stands, there is currently no known attack against it.

3.5 Encryption of traffic by the card itself

If we use a smartcard only to negotiate session keys via IKE, an IPSec connection can stay open after a user has removed the card: as long as the session key does not expire there is no need for an SKEYID calculation. More secure behaviour would be that the host can communicate through IPSec connection only when the card is accessible for the host. From a user perspective this also is more intuitive: the user can be assured that no IPSec communication is possible when the smartcard is not inserted.

In this model we require that:

1. The receiving host can verify that a received packet was sent when the sending host had access to the smartcard (so it can discard packet for which this does not hold).

2. The receiving host cannot decrypt received data packets without access to the smartcard.

One might try to forfill these requirements by bringing down the IPSec connection when the smartcard is removed. To reestablish the connection IKE would need to renegotiate session keys, which it cannot do without the card. In general this feature is a good addition to basic smartcard enabled SKEYID calculation: it allows the user to regard its smartcard as an ignition key. Card inserted means connectivity, card not inserted means no connectivity.

However, this method does not strictly conform to our requirements. The sending host can send packets that will be accept by the recipient even when it does not have access to the smartcard, it simply must not bring down the connection when the card is removed. Likewise the host can keep decrypting data it receives. The peer host is dependent on 'decent' behaviour by its communicating partner.

To enforce the requirements a more secure method has to be employed. Just negotiating session keys with the card does not suffice, the encryption and decryption of the data itself has to depend on the card. In the next chapters we will discuss how this might be accomplished with the help of a concept called *remotely keyed encryption* (RKE).

4 Remotely Keyed Encryption

Many security-relevant applications store secret keys on a tamper-resistant device, a *smart card.* Protecting the valuable keys is the card's main purpose. Although in recent years some interesting cryptographic [Weis97] and many very dangerous hardware attacks [WTK97] have been mounteded, smart cards provide much higher security than other storage systems.

Two problems remain however with regards to using this device for bulk encryption.

Performance Problem

The first lies in the physical limitations of smart cards which make them typically slow. Fortunately new cryptographic protocols make fast encryption on a smart card supported system possible. Small JAVA devices are too slow to provide an acceptable bandwidth. Fortunately, remotely keyed encryption schemes are designed to allow "High-Bandwidth Encryption with Low-Bandwidth Smart cards" [Blaz96]. (This is accomplished with the help of a fast but untrusted host.)

Export Restrictions

The second major problem is posed by the restrictions on encryption hardware. Our legally exported JAVA rings do offer native support for encryption. Since far fewer restrictions regarding authentication or signature tools exist, protocols have been developed by Lucks [Luck97] and Weis [Weis99b], that

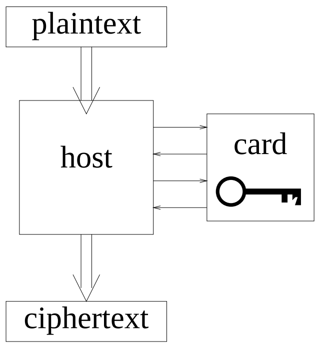

Figure 3: Remotely Keyed Encryption

are well suited to use a built-in SHA-1 hash function (from a legal standpoint a hash function is NOT an encryption function!) [LuWe99a] for bandwidth-greedy cryptographic services like file encryption.

Main Ideas

The notion of *remotely keyed encryption* is due to Blaze [Blaz96]. A *remotely keyed encryption scheme* (RKES) distributes the computational burden for a block cipher with large blocks between two parties, a *host* and a *card*. We think of the host as being a computer under the risk of being taken over by an adversary, while the card can be a smart card, protecting the secret key.

The host knows plaintext and ciphertext, but only the card is entrusted with the key.

An RKES consists of two protocols: the *encryption protocol* and the *decryption protocol*. Given a B-bit input, either to encrypt or to decrypt, such a protocol runs like this: The host sends a *challenge value* to the card, depending on the input, and the card replies with a *response value*, depending on both the challenge value and the key.

5 Security Model

One approach with very high practical implication is the analysis of different attack scenarios.

5.1 Attacks on Stored Data

The one of the biggest problem for designing secure systems is the key management. In the Internet age there is no "safe place" to store the secret keys, for a connected computer.

If someone uses a Windows system there does not seem to be even a theoretical chance of avoiding attacks over the network. Besides the known attacks with "Trojan Horses" (e.g. Back Orifice), which are very hard to prevent, since we have no open sources, not even a "randomized" swapping of memory to the hard disk can be prevented. If key bits or even only intermediate values can be located on the swap areas of the hard disk, the security of the system may be compromised.

Windows Memory Lock [Gutm99]

Win16: No security

Win95: `VirtualLock()` does nothing

WinNT: `VirtualLock()` data is still swapped

5.2 Attack Scenarios

In this section we discuss our standpoint, that in open systems, smart card supported systems represents the only practicable solution for sensitive data.

We want to discuss two main attack scenarios. In an *offline-attack* the attacker gains physical control of the harddisk. In the an *online-attack* the attacker takes control of the host system and is able to communicate with the smart card.

Security Problems of a Software Solution

A software-only solution is not secure against both kind of attacks.

Offline Attack:

If the attacker gains control of the hard disk (e.g. steals the notebook) she can try to perform a *dictionary attack*. Most humans use passwords with a very poor entropy.

In additional she can search "random looking" data in the swap areas. Note that this strategy of "Playing Hide and Seek with Stored Keys" was suggested by Shamir and Someren [ShSo99] and has helped to find the Microsoft "_NSAKEY".

Figure 4: Looking for random data in a swap file [ShSo99]

Online Attacks:

If an attacker can take control of the host during the encryption she can read the secret key.

Security of Smart Card Supported Solutions

The security of smart card supported systems is much higher.

Offline Attacks:

Given a sufficiently high entropy and length of the secret key in the smart card, a Brute Force Attack seems to be infeasible. An attacker has to steal the hard disk AND the smart card AND crack the smart card PIN.

Online Attacks:

The security proves of our protocols show that an attacker who has control of the host system can only read files which are decrypted while she is in control.

5.3 A 3 Line Security Model

This leads us to a 3 line security model.

- **Hosts** cannot be secure.
- **Smart cards** are pretty secure.
- $\Longrightarrow$ The secret key must NEVER leave the card.

Smart cards are also user-friendly. They provide a ”The key in your hand” feeling and you can ”carry” your secret key with you.

6 Random Mapping Based Protocols

The theoretical publications of Stefan Lucks ([Luck96], [Luck97]) have stimulated the use of Luby/Rackoff construction in the context of smart card supported encryption protocols. Compared to other approaches these protocols are based on a strong mathematical model and we can *prove* the margins of security.

Further in [LuWe99a] Lucks and the author have shown implementation of encryption protocols with non-encrypting smart cards. Has a big practical impact since for non-encrypting smartcards there are much less resrictions and finally they are much cheaper than cards with "strong cryptography".

The RaMaRK Encryption scheme

In this section, we describe the *Random Mapping based Remotely Keyed (RaMaRK) Encryption scheme*, which uses several independent instances of a *fixed size random mapping* $f : \{0,1\}^b \longrightarrow \{0,1\}^b$. The scheme is provably secure if its building blocks are, i.e., it satisfies the requirements (i)–(iii) above, see [Luck97]. Note that b must be large enough to make performing close to $2^{b/2}$ encryptions infeasible. We recommend to choose

$$b \geq 160.$$

By "$\oplus$" we denote the bit-wise XOR, though mathematically any group operation would do the job as well.
We use three building blocks:

1. Key-dependent (pseudo-)random mappings

$$f_i : \{0,1\}^b \longrightarrow \{0,1\}^b.$$

2. A hash function

$$H : \{0,1\}^* \longrightarrow \{0,1\}^b.$$

   H has to be *collision resistant.*

3. A pseudorandom bit generator (i.e. a "stream cipher")

$$S : \{0,1\}^b \longrightarrow \{0,1\}^*.$$

   If the seed $s \in \{0,1\}^b$ is randomly chosen, the bits produced by $S(s)$ have to be indistinguishable from randomly generated bits.

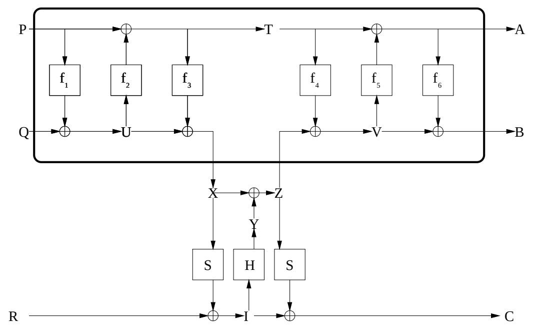

Figure 5: The RaMaRK encryption protocol

> In addition to pseudorandomness, the following property is needed: If s is secret and attackers choose $t_1, t_2, \ldots \in \{0,1\}^b$ with $t_i \neq t_j$ for $i \neq j$ and receive outputs $S(s \oplus t_1)$, $S(s \oplus t_2)$, $\ldots$, it has to be infeasible for the attackers to distinguish these outputs from independently generated random bit strings of the same size. Hence, such a construction behaves like a random mapping $\{0,1\}^b \longrightarrow \{0,1\}^{B-2b}$, though it is actually a pseudorandom one, depending on the secret s.

Based on these building blocks, we realize a remotely keyed encryption scheme to encrypt blocks of any size $B \geq 3b$, see figure 5.
We represent the plaintext by (P,Q,R) and the ciphertext by (A,B,C), where $(P,Q,R),(A,B,C) \in \{0,1\}^b \times \{0,1\}^b \times \{0,1\}^{B-2b}$. For the protocol description we also consider intermediate values $T,U,V,X,Y,Z \in \{0,1\}^b$ and $I \in \{0,1\}^{B-2b}$. The encryption protocol works as follows:

1. Given the plaintext (P,Q,R), the host sends P and Q to the card.
2. The card computes $U = f_1(P) \oplus Q$ and $T = f_2(U) \oplus P$, and sends $X = f_3(T) \oplus U$ to the host.
3. The host computes $I = S(X) \oplus R$ and $Y = H(I)$, sends $Z = X \oplus Y$ to the card, and computes $C = S(Z) \oplus I$.
4. The card computes $V = f_4(T) \oplus Z$, and sends the two values $A = f_5(V) \oplus T$ and $B = f_6(A) \oplus V$ to the host.

The decryption protocol is very similar (s. Fig. 5 or [Luck97]).

If the block size B of the cipher it realizes is not too small compared to the parameter b, the RaMaRK scheme is efficient. The card itself operates on $2 \cdot b$ bit data blocks, and both $3 \cdot b$ bit of information enter and leave the card.

6.1 The Security of RaMaRK

Lucks [Luck97] pointed out some weaknesses of Blaze's scheme and gave formal requirements for the security of RKESs:

(i) Forgery security: If the adversary has controlled the host for $q-1$ interactions, she cannot produce q plaintext/ciphertext pairs.

(ii) Inversion security: An adversary with (legitimate) access to encryption must not be able to decrypt and vice versa.

(iii) Pseudorandomness: The encryption function should behave pseudorandomly for someone without access to the card, nor knowledge of the secret key.

While requirements (i) and (ii) restrict the abilities of an adversary with access to the smart card, requirement (iii) is only valid for *outsider adversaries* without access to the card. If an adversary could compute forgeries or run inversion attacks, she could easily distinguish the encryption function from a random one.

6.2 Extended Security Model

Blaze, Feigenbaum (AT&T) and Naor (Weizmann Institut) [BFN98] published recently a paper on the EUROCRYPT'98 which has showed a new formal model for RKES, found a problem in the RaMaRK protocol and suggested a new RKES, that fulfills the new security model.

BFN Model of Pseudorandomness of a RKES

It is theoretically desirable that a cryptographic primitive always appears to behave randomly to everyone without access to the key. In any RKES, the amount of communication between host and card should be less than the input length, otherwise the card could just do the complete encryption on its own. Since (at least) a part of the input is not handled by the smart card, and, for the same reasons, (at least) a part of the output is generated by the host, an insider adversary can easily decide that the output generated by herself is not random.

Blaze, Feigenbaum, and Naor [BFN98] found another way to define the pseudorandomness of RKESs. Their formal definition is quite complicated. It is based on the following scenario:

Adversary A is gains direct access to the card *for a certain amount of time* and makes a fixed number of interactions with the card. Ones A has lost direct access to the card, the encryption function should appear to behave randomly, even to A.

Security Problems of the RaMaRK scheme

Regarding the RaMaRK scheme they pointed out that an adversary A who has had access to the card and lost the access again, can later choose special plaintexts where A can predict *a part of* the ciphertext. This makes it easy for A to distinguish between RaMaRK encryption and encrypting randomly.

The intermediate value X depends only on the (P, Q)-part of the plaintext, and the encryption of the R-part depends only on X. If A chooses a plaintext (P, Q, R), having participated before in the encryption of (P, Q, R'), with $R \neq R'$, the adversary A can predict the C-part of the ciphertext, but not the P nor the Q part, corresponding to (P, Q, R) on her own.

Thus, according to the definition of [BFN98], the RaMaRK scheme is not pseudorandom.

6.3 "Decryption" of the Ciphertexts

In [BFN98]Blaze, Naor and Feigenbaum mentioned another strong concern. They pointed out that in some cases it may be feasible to decrypt parts of the ciphertext after an online attack. It was shown in [Weis99b] that this statement does not apply.

The authors of [BFN98] pointed that there is a possibility to attack files with the same $2b$ bit header.

> "However, because the encryption key depends only on the first two plaintext blocks, an arbitrarily large set of messages all of which start with the same two blocks will always be encrypted with the same key. This is not a hypothetical situation: A set of files in a computer file system, for example, might always start with the same few bytes of structural information."

The above describes a *known plaintext* distinguishing attack, that is actually feasible. The authors of [BFN98] continue:

> "An adversary that controls the host during the encryption or decryption of *one* file in such a set could subsequently decrypt the encryption of *any* file in the set."

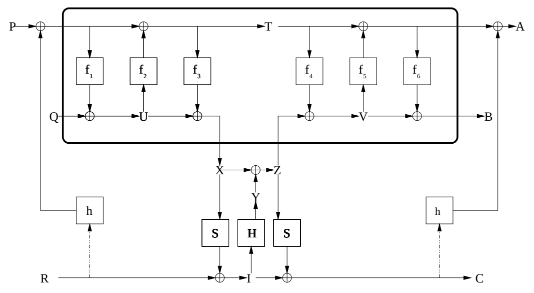

Figure 6: Improved RaMaRK [Weis99b]

We argue that this attack ist *not* feasible. Note that the second intermediate key Z (resp. X for decryption) depends on all bits of the plaintext (P, Q, R) (resp. ciphertext (A, B, C)).

$$Z = X \oplus Y = X \oplus H(I) = X \oplus H\Big(\boxed{\text{R}} \oplus S(X)\Big)$$

Thus the knowledge of the intermediate value X (resp. Z) is not sufficient for a decryption of *any* file of the mentioned set of files.

On the other hand it is a not satisfactory cryptographic property that an attacker can peel off one of the two stream cipher encryptions if she knows the intermediate key X.

$$C = I \oplus S(Z) = R \oplus S(X) \oplus S(Z)$$

7 The Improved RaMaRK protocol

Because of the reasons discussed in the last section the author has suggested a slight modification of the protocol on the host side [Weis99b].

Protocol Modification

We want to make sure that also the intermediate keys X and Y depend on every plaintext bit. Instead of P and Q we submit

where h is a cryptographic hash function.

The *Improved RaMaRK scheme* is interface-compatible with the unmodified RaMaRK scheme. So no hardware modifications to the smart card are necessary.

7.1 Characteristics and Limitations

If we choose a standard hash function with 160-bit output, a *known plaintext attack* against the pseudorandom property seems to be infeasible.

According to Lucks, a *chosen plaintext attack* in the BFN scenario to distinguish the output of the protocol from a random output is still feasible. So even the improved RaMaRK scheme does not meet the stronger security model of [BFN98].

Further more it is not possible to peel off one stream cipher encryption as discussed in the last section.

The modification requires two expensive hash function calls for the big block B. We do not expect this to cause a problem for most applications since the main bottleneck seems to be the communication with the card.

8 Accelerated Remotely Keyed Encryption

Blaze, Feigenbaum and Naor have published at Eurocrypt '98 a stronger security model for Remotely Keyed Encryption schemes [BFN98]. They also present a new protocol in which an attacker who gets the control over the host system for a certain amount of time can not get a significant advantage for the time after she lost control. Further the pointed out some weaknesses in the RaMaRK protocol [Luck97]. Some of these critics have been fixed by [Weis99b], but these modifications do not fulfill the stronger security model. Stefan Lucks has improved the BFN protocol on the Fast Software Encryption 1999 by presenting the Accelerated Remotely Keyed Encryption Scheme (ARK). To avoid confusion we use the same notations as in [Luck99] where ever it is possible.

8.1 Building Blocks and Security Assumptions

In [Luck99] was proven that the security of the ARK scheme is closely related to the security of the building blocks. In this section we describe the requirements for these blocks. We try to make these requirements as weak as possible. This strategy will provide a bigger margin of security.

8.2 Security Parameter

By a we denote the block size of the block cipher E (usually 64 or 128 bit). Let b be the output size of the hash function H (usually 128 or 160 bit). These numbers are very important security parameters. Especially in most practical scenarios a 64-bit block cipher such as Triple-DES seems to be not appropriate.

8.3 building blocks

We use the following building blocks for the ARK protocol.

- an a-bit block cipher $E_K : \{0,1\}^a \rightarrow \{0,1\}^a$
  (e.g. AES:$\{0,1\}^{128} \rightarrow \{0,1\}^{128}$, Rijndael, Twofish, DEAL/SK)
- a family of pseudorandom functions $F_K\{0,1\}^b \longrightarrow \{0,1\}^a$
  (e.g. AES based CBC-MAC: $\{0,1\}^{160} \rightarrow \{0,1\}^{128}$)
- a hash function $H : \{0,1\}^* \longrightarrow \{0,1\}^b$
  (e.g. SHA-1, RIPE-MD160: $\{0,1\}^* \longrightarrow \{0,1\}^{160}$)
- a length-preserving stream cipher $S : \{0,1\}^* \longrightarrow \{0,1\}^*$,
  depending on an a-bit key.
  (e.g. ARCfour, AES/OFB)

8.4 Security Assumptions

The security assumptions are

1. E_K is a random permutation over $\{0,1\}^a$, and for $K \neq K'$ the permutations E_K and $E_{K'}$ are independent.
2. $F_K\{0,1\}^b \longrightarrow \{0,1\}^a$, is a random function, i.e., a table of 2^b random values in $\{0,1\}^a$. Similarly to above, two random functions depending on independently chosen keys are assumed to be independent.
3. H is be *collision resistant*, i.e., the adversary unable to find a pair $(V, V') \in \{0,1\}^*$ with $V \neq V'$ and $H(V) \neq H(V')$ if $V \neq V'$.
4. S_K is a length-preserving stream cipher, depending on a key $K \in \{0,1\}^a$. I.e., for every number n, every plaintext $T \in \{0,1\}^n$, every set of keys $L = \{K_1, \ldots, K_r\} \subseteq \{0,1\}^a$ and every key $K \in \{0,1\}^a$, $K \notin L$, the value $S_K(T) \in \{0,1\}^n$ is a random value, independent of T, $S_{K_1}(T)$, …, $S_{K_n}(T)$.

9 The ARK encryption scheme

In this section we describe the Accelerated Remotely Keyed (ARK) Encryption Scheme.

9.1 Notation

We use two pseudorandom permutations

$$E_1, E_2 \quad \text{over} \quad \{0,1\}^a$$

and two pseudorandom functions

$$F_1, F_2 : \{0,1\}^b \longrightarrow \{0,1\}^a$$

The encryption function mapes any B-bit plaintexts ta B-bit ciphertext. The scheme can use any blocksize B with $B \geq a$.
We represent the plaintext by

$$(P, Q) \quad \text{with} \quad P \in \{0,1\}^a \quad \text{and} \quad Q \in \{0,1\}^{B-a}.$$

Similarly we represent the ciphertext by

$$(C, D) \quad \text{with} \quad C \in \{0,1\}^a \quad \text{and} \quad D \in \{0,1\}^{B-a}.$$

For the protocol description, we also consider intermediate values $X, Z \in \{0,1\}^b$ and $Y \in \{0,1\}^a$. The encryption protocol works as follows:

9.2 Encryption Protocol

Given the plaintext (P, Q)

1. The host sends P and $X := H(Q)$ to the card.
2. The card responds $Y := E_1(P) \oplus F_1(X)$.
3. The host computes $D := S_Y(C)$.
4. The host sends $Z := H(D)$ to the card.
5. The card responds $C := E_2(Y \oplus F_2(Z))$.

The decryption protocol is very similar (s. Fig. 7 or [Luck99]).

9.3 Separation of En- and Decryption

Note that if E_1 and E_2 are independent or if F_1 and F_2 are indepenent, the decryption protocol cannot be used for encryption. This describes some kind of authentication mechanism which would usually require heavy-weight public-key cryptography. We may produce cards which can only be used for encryption or only for decryption – even if the card holder tries to misuse the cards (but cannot break their tamper-resistance). E.g., we can prevent a receiver from faking a pay-tv program.

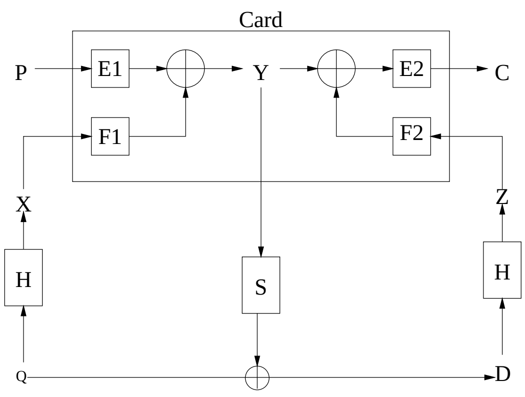

Figure 7: Accelerated Remotely Keyed Encryption [Luck99]

10 Application of RKE for IP packet encryption

RKE encryption of IP traffic is not the same as encryption of a file system or video stream, it has its own properties and requirements. For one, the block size is not fixed. IP traffic is handled a packet at a time, so every encryption block typically contains exactly one IP packet. Only when multiple packets wait in the transmit queue the block may consist of multiple blocks. Else applications on both hosts could experience unacceptable latencies or even dead lock. Clearly we would not want to make the encryption block size smaller than the IP packet since: it does not give increased security but costs extra overhead.

In most cases IP packets will be smaller than the maximum packet size of the underlying datalink in order to avoid fragmentation. For Ethernet LAN environments this results in a maximum block size of less than 1500 bytes.

A second consideration is that internet traffic also puts strong requirements on latency. The round trip time of the encrypted connection minimally is four times the time the smartcard needs to process a single block.

We have performed some timing measurements with the iButton (see paragraph 12.4). Unfortunately we had to conclude that the current iButton revision is prohibitively slow for remotely keyed encryption of internet traffic.

11 Restricting Usage of the iButton

A common principle is to base authentication on both something we own (in this case the iButton) *and* something we know (i.e. a pass phrase). This ensures that simply stealing the token will not allow an adversary access to the protected services.

11.1 The ISO7816 external authenticate command

Traditionally the 'something we know' part has been implemented with an 'enter PIN' command (called 'external authenticate' in ISO7816-4) provided by the hardware token: initially the token bars access to its protected commands, in our case the calcSkeyID command. The application then issues the enter PIN command with a PIN given by the user. The token checks the PIN. If it is correct the token changes its volatile state, allowing access to the protected commands. After a reset the state will be 'bar access' again. In case of an incorrect PIN the token decreases[3] a non volatile retry counter, if the counter reaches zero the token will not accept enter PIN commands anymore. Most implementations allow this retry counter to be reset after an administrative PIN is entered. Typically this PIN is only know by the issuer of the token, not the end user.

There are several drawbacks to the use of the PIN feature:

1. Facilitates an offline attack. An adversary can try PINs (up till the retry count limit) without anyone noticing since the token gives immediate feedback about the correctness of the PIN.

2. Once a PIN is blocked due to too many failed enter PIN attempts, the user either has to obtain a new token or (physically) go to the token issuer to have the PIN unblocked again.

3. Many implementations of PIN controlled access in smart cards have been shown to contain flaws, that allow one to deduce the correct PIN or bypass the retry counter [SHB97].

4. The statefulness of the method might allow unauthorized applications access to the protected commands. Example scenario: the user starts a trusted application and gives it the PIN of the token. The application enters the PIN, but crashes or gets killed before it can reset the token. Now any application that can access the token immediately has access to the protected commands, without the user even knowing.

[3]actually any decent PIN implementation decreases the counter before the check and increase the counter again if it is OK. This prevents an attack where one measures the power consumption of the card to see if it tries to write to nonvolatile memory (power consumption rises directly before the write). One then simply prevents the write operation (and thereby the counter decrement) by terminating power to the card.

To counter this, the button might request the PIN on every invocation of a protected command. However, the ISO7816-4 standard does not provide this, so few, if any, smartcard implement such a feature.

11.2 A naive alternative

We would like to have a method that answers better to the actual goal: shielding the usage of the correct authentication key rather than protecting access to the calcSkeyID command.

An initial effort to provide this, that also has been implemented in the iButton applet, is the following construction: the PIN P shall not be used as a password but as a Key Encrypting Key (KEK) for the authentication key K. The smartcard contains the encrypted authentication key Q. So the SKEYID calculation becomes:

$$SKEYID = prf(E_P^{-1}(Q), N_i|N_r)$$

The Encryption algorithm of our choice was the ever favorite exclusive OR function, firstly because it's fast, easy and available everywhere (unencumbered by crypto export restrictions). But another desirable feature is that it allows to trivially change the key without having to decrypt the cipher text.

A clear advantage of this scheme is that the iButton does not contain the cleartext value of either the secret key or the PIN. Also the implementation contains only a single execution path, it does not have to decide whether the given PIN is correct. In effect this counters the attacks on PIN deduction from subliminal paths. Also if an adversary manages to read the contents of the button, for example by microprobing, she has not yet obtained the key. Only after a successful PIN guessing attack the key becomes available.

The scheme is incompatible with the retry counter feature of the ISO7816 PIN command: since the button does not know the correct PIN it cannot decrease the counter if the PIN is not correct. The purpose of the retry counter is to thwart attempts to obtain the PIN by guessing them until the correct one is found. If we can ensure that an attacker can only verify if a PIN is correct if he has to contact one of the IPSec hosts each try we do not need a retry counter in the smartcard: the IPSec implementation can detect the PIN guessing attempt and deal with it in a much more flexible way than a smartcard could (for example by temporarily blocking the account, notifying a security officer, etc.).

However the presented scheme is *not* secure against offline PIN guessing. Obviously if an adversary knows a valid triplet {Nonce-I, Nonce-R, SKEYID}

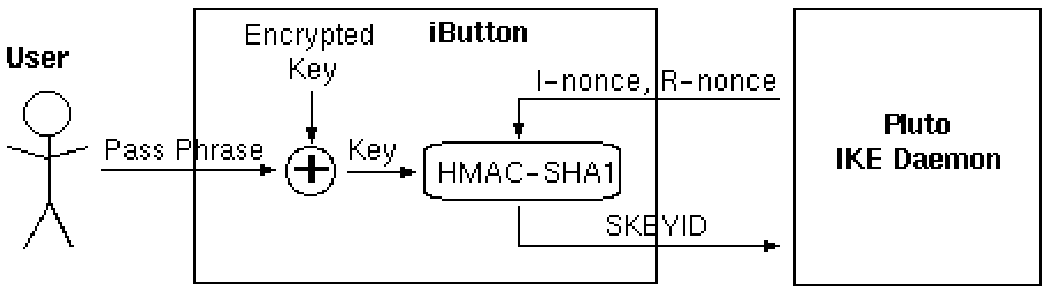

Figure 8: Pass phrase keyed calculation of SKEYID

he can verify a PIN by sending it and the nonces to the card and checking whether the card returns the same SKEYID. The chances for an adversary to know such a triplet may be very slim, but much more easily obtainable information will do to: the adversary does not actually need to know the SKEYID he just has to verify whether it is the correct one. Coincidentally the I-digest and R-digest in the IKE exchange have been designed to ensure the peer party has the correct SKEYID.

An adversary can abuse this feature to check on 'candidate' SKEYIDs. The first half of this attack is online: he provokes (or waits for) one of the parties to initiate a key exchange. He then answers this initiation an takes the role of the responder. He records all exchanged data and cuts off the communication at the last step (sending the R-digest) since he cannot send the correct R-digest anyway. He now has all parameters necessary for the calculation of the I-digest, except two: the SKEYID and the ID_{i1}. Now he can perform the second, offline, half of the attack: guess a PIN, let the smartcard generate the candidate SKEYID and verify it. For verification he calculates all derived SKEYIDs from the candidate SKEYID and all obtained parameters. Then he decrypts ID_{i1}, calculates the I-digest and compares it to the (decrypted) I-digest received from the initiator. If the match the SKEYID is correct and consequently the PIN too.

This attack enables the adversary to try as many PINs as necessary without having to contact one of the parties more than once, thereby circumventing any online retry counters.

11.3 A potential improvement

The problem mentioned above is caused by the fact that an adversary can completely specify the input to the SKEYID calculation on the smartcard. Therefore she can 'replay' the same IKE negotiation over and over again with different PINs.

In order to prevent such a replay we suggest the generation of an extra nonce by the smartcard, to be incorporated in the SKEYID calculation. The SKEYID then becomes:

$$SKEYID = prf(pre - sharedkey, N_{ih}|N_{is}|N_{rh}|N_{rs})$$

where N_{ih} is the initiators host nonce, N_{is} is the nonce generated by the smartcard of the initiator and N_{rh} and N_{rs} are the respective responder nonces. This scheme is compatible with the existing IKE protocol, we can simply let $\mathrm{N}_i := \mathrm{N}_{ih} \mid \mathrm{N}_{is}$ and $\mathrm{N}_r := \mathrm{N}_{rh} \mid \mathrm{N}_{rs}$.

However we have not yet performed a security analysis for this construct, further investigation is recommended.

12 Implementation

Method 1 (smartcard file system) and 2 (SKEYID calculation) have been implemented for FreeS/WAN 1.3 (backporting should be simple). Because of the many open issues surrounding the RKE approach, no attempt has been made to implement it. All written code is has been placed under the General Public License.

The implementation has been split into three parts: the iButton applet, iButton host code and the FreeS/WAN integration code. The iButton applet has been developed with the iButton IDE 1.01 running on the Blackdown JDK1.2.2pre4.

The host code has been written in C/C++ using the provided C-API. Having to run a complete Java Runtime Environment for a single SKEYID calculation is a bit overkill.

12.1 The FreeS/WAN integration code

Luckily FreeS/WAN contains a single point where the SKEYID is derived from the long term shared secret and the initiator and responder nonces: the function skeyid_preshared() in ipsec_doi.c of the Pluto daemon. The FreeS/WAN integration has been realized be patching this file. It has been designed to support both method 1 and 2 and to impose as little limitations on where the shared secret may be retrieved from or how the SKEYID might have been generated. Specifically the patch is completely free of any user interaction code. The patched Pluto daemon checks whether the shared secret defined in /etc/ipsec.secrets starts with 'exec '. If so, the rest of the secret is assumed to be a command Pluto has to execute to obtain either the real shared secret or the calculated SKEYID, which one is up to the

CLA	INS	method	description
BB	0	setID	initializes the button with the shared secret

BB	1	getParameter	retrieves a parameter:
			P2 = 0: connection name
			P2 = 1: shared secret (debug version only!)

BB	2	calcSkeyid	calculates SKEYID from given challenge.
			P2 = 0: no password given
			P2 = 1: use given password

BB	3	changePassword	change password, i.e. offset shared secret with given value

Table 1: instructions implemented in the iButton applet

discretion of the command. Pluto passes the necessary parameters, such as local and remote IP address, initiator and responder nonce, connection name and HMAC hash type, to the command by setting environment variables. The command returns the result by printing a single line to its standard output.

The implementors of the executed command are completely free in the way they interact with the user and hardware token. In the future a framework for this may be developed, that also includes a more appealing alternative interface to other parts of FreeS/WAN, than 'vi /etc/ipsec.conf' and the /usr/local/sbin/ipsec command.

12.2 The iButton Applet

The IPSecID applet provides the smartcard services needed for SKEYID calculation based authentication. It implements the 4 instructions shown in table 1. The described implementation includes the insecure alternative PIN feature described in paragraph 11.2.

During the applet development we observed that the SHA1 calculation on the iButton is not fully conforming [SHA1]. The digest the button generates has the wrong byte order, that is the bytes are stored in order 32107654..... instead of 01234567...... iButton support confirmed this behaviour, but did not give a reason or indication that it will be corrected in a next iButton firmware release. Of course, a workaround was simple. Other than that, we did not encounter any difficulties or problems.

12.3 Host code

The iButton IDE version we used did not include source code for the Linux C-API, but the SPARC Solaris version compiled without problem on Linux. With it we wrote some small C programs for setting the shared secret in the iButton and for calculation of the SKEYID. They consist of little more than a wrapper around the iButton instructions. The 'calcSkeyid' program conforms to the interface described in paragraph 12.1, so it can be used in combination with FreeS/WAN. It does not perform any user interaction: if the iButton is not inserted the user is not notified, but the IKE negotiation directly fails instead. Clearly for actual deployment a more user friendly version is desirable.

12.4 Some timing measurements

In order to get some indication of the speed of the button and its API, some timing tests have been performed. The test platform was an AMD K7 Athlon 500MHz system containing 128MB with RedHat Linux 6.1 installed. The iButton had firmware release 1.11.006 and contained only the IPSecID applet, in its first slot. The button was connected to the computer with the serial blue dot iButton receptor. The results of six test runs is show in table 2.

First button communication initialization ('initJIB') and close down ('cleanup') were measured. The initialization turned out to be slower than we expected: it takes almost 0.3 seconds. The cleanup is much cheaper and completes within a millisecond.

Next we timed a very simple instruction: a 'ping' instruction, which simply echos back the data (if any) sent as a parameter. A ping without data is the simplest instruction an iButton can execute, it is an empty method. Yet it takes 0.67 seconds to complete. When we sent some data along (1 bytes sufficed) this increase to 1.16 seconds. These values are much higher than we expected based on experience with conventional smartcards[4].

We also timed two of the instructions in the IPSecID applet: getParameter and calcSkeyID. The getParameter performs in the same range as the ping command, not surprisingly as it does little more than returning a byte array.

The calcSkeyID instruction is much more complex as it needs to perform two SHA1 hash calculations. Apparently this is a rather heavy job for the iButton, it takes almost 3.5 seconds for the completion of the instruction. The

[4]a typical Chipper or ChipKnip card can handle over 20 'select file' commands in a second, ie. the 'ping time' is less than 50 milli seconds.

action	1	2	3	4	5	6	average
initJIB	0.298	0.293	0.300	0.296	0.298	0.296	0.296
cleanup	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Ping	0.670	0.670	0.670	0.670	0.670	0.670	0.670
Ping w. data	1.160	1.160	1.160	1.160	1.160	1.160	1.160
getParameter	0.96	0.97	0.97	0.96	0.96	0.97	0.97
calcSkeyID	3.47	3.44	3.44	3.45	3.44	3.45	3.45
calcSkeyID w. pass	3.45	3.45	3.45	3.46	3.45	3.45	3.45

Table 2: time measurements (in seconds) of 6 runs of iButton operations

variant of the instruction which takes a 64 byte password as extra parameter is not significantly slower.

According to iButton support <jibsupport@dalsemi.com> "1.5-2.0 seconds per hit are normal.".

13 Conclusions

Storing long term secrets for IPSec in a hardware token like the iButton can considerably improve the security of the VPN. The Internet Key Exchange (IKE) permits storage of the long term keys in a safer place than on hard disk without changes to the protocol.

Remotely Keyed Encryption is a powerful concept for smartcard enabled encryption that copes with one of the major problems for application of smartcard in bulk encryption: the low bandwidth of the card. Fast and secure RKE protocols are available that work around another, political problem: export restrictions on smartcards that implement strong encryption.

The iButtons Java Card technology gives smartcard application developers much more freedom and flexibility over conventional smartcard technology. It enabled us to quickly build a small smartcard applet for secure authentication of IPSec key exchanges, something not doable with conventional cards.

The iButton is a relatively slow device: it takes more than 3 seconds for an HMAC calculation. This prohibits the application of Remotely Keyed Encryption protocols, because the resulting latency of the encrypted IP traffic will be too high to be usable.

14 Future Work

The proposed methods for storage of IPSec's long term keys in iButtons can be extended to other hardware tokens that support the SHA1 algorithm. For Java Card compliant tokens the adaption should be trivial. Another extension is addition of support for IKE based on public keys. A method 1 implementation is simple, for method 2 one requires public key operations in the token, so this will require export restricted tokens.

Furthermore, it should be easy to apply the presented methods and implementations to other IPSec implementations, such as NIST's Linux IPSec or Free- and NetBSD's IPSec implementation.

Of course a feasible method 3 solution remains an interesting subject too. This requires a token that is fast enough to keep the latency within reasonable limit, a task the current iButton revision cannot handle.

Lastly we have implemented but the basics of iButton enabled key negotiation. For actual use, a good user interface has to be built. Currently FreeS/WAN itself is lacking a user friendly interface too.

References

[BFN98] Blaze, M., Feigenbaum, J., and Naor, M., "A Formal Treatment of Remotely Keyed Encryption (Extended Abstract)", Eurocrypt '98, Springer LNCS 1403, 1998.

[Blaz96] Blaze, M. "High-Bandwidth Encryption with Low-Bandwidth Smartcards", Fast Software Encryption (ed. D. Gollmann), Springer LNCS 1039, 33-40, 1996.

[Chip] Website Chipper Netherlands:
`http://www.chipper.nl/standvanzaken/cijfers.html`

[Dallas] Website Dallas Semiconductor iButton:
`http://www.ibutton.com/ibuttons/java.html`

[DH] Diffie, W., and Hellman M., "New Directions in Cryptography", IEEE Transactions on Information Theory, V. IT-22, n. 6, June 1977.

[ESP] Kent, S., and Atkinson, R., "IP Encapsulating Security Payload (ESP)", RFC2406, November 1998.

[FeSch00] Niels Ferguson and Bruce Schneier, "A Cryptographic Evaluation of IPSec", January 2000.

[FreeSWAN] Website FreeS/WAN: `http://www.freeswan.org`

[Gutm99] Gutmann, P., "Windows Memory Lock", godzilla crypto tutorial, 1999.
`http://www.cs.auckland.ac.nz/~pgut001/tutorial/`

[HMAC] Krawczyk, H., Bellare, M., and Canetti, R. "HMAC: Keyed-Hashing for Message Authentication", RFC2104, February 1997.

[HMAC-SHA] Madson, C., and R. Glenn, “The Use of HMAC-SHA-1-96 within ESP and AH", RFC 2404, November 1998.

[IKE] Dan Harkins and Dave Carrel, “The Internet Key Exchange (IKE)”, draft-ietf-ipsec-ike-01.txt, May 1999.

[Ipay] Website Interpay: `http://www.interpay.nl/ws?pg=kern-toon.html`

[IPSec] Website IETF IPSec working group: `http://www.ietf.org/html.charters/ipsec-charter.html`

[IHR98] Itoi, N., Honeyman, P. and J. Rees, “SCFS: A UNIX Filesystem for Smartcards”, December 1998. {Proc. USENIX Workshop on Smartcard Technology, Chicago (May 1999).}

[Kiv99] Kivinen, T, “Fixing IKE Phase 1 & 2 Authentication HASHes”, draft-ietf-ipsec-ike-hash-revised-01.txt, March 1999.

[Luck96] Lucks, S. “Faster Lubby-Rackoff ciphers”, Fast Software Encryption, Springer LNCS 1039, 1996.

[Luck97] Lucks, S., “On the Security of Remotely Keyed Encryption”, Fast Software Encryption, (ed. E. Biham), Springer LNCS, 1997.

[Luck99] Lucks, S., “Accelerated Remotely Keyed Encryption”, Fast Software Encryption, Springer LNCS, 1999.

[LuWe99a] Lucks, S., and R. Weis, “Remotely Keyed Encryption Using Non-Encrypting Smart Cards”, USENIX Workshop on Smartcard Technology, Chicago, May 1999.

[SHA1] NIST, FIPS PUB 180-1: “Secure Hash Standard”, April 1995.

[SHB97] Squire, B., Hemel, T., and Bakker, B., “Smartcard Hacking”, Hacking in Progress 97, Almere, 1997.

[ShSo99] Shamir, A, and van Someren, N., “Playing ’hide and seek’ with stored keys”, Financial Cryptography ’99, Anguilla, BWI, 1999

[Sun00] `http://www.sun.com/products/javacard/`

[WeBo00] Weis, R. Bogk, A., “Secure High Speed Video Encryption”, convergence integrated media GmbH (Berlin, San Francisco, Amsterdam), CEBIT 2000, Hannover, 2000.

[Weis97] Weis, R. “Combined Cryptoanalytic Attacks against Signature- and Encryption schemes”, (in German), A la Card aktuell 23/97, 1997, pp. 279ff.

[Weis99a] Weis, R., “Crypto Hacking Export Restrictions”, Chaos Communication Camp, Berlin, 1999

[Weis99b] Weis, R., “A Protocol Improvement for High-Bandwidth Encryption Using Non-Encrypting Smart Cards”, IFIP TC-11, Working Groups 11.1 and 11.2, 7th Annual Working Conference on Information Security Management & Small Systems Security, Amsterdam, 1999.

[Weis00] Weis, R., “A Trivial Host Card Encryption Protocol”, February 2000, to be published

[WTK97] Weis, R., Kuhn, M., and Tron, ”Hacking Chipcards”, CCC 1997, Hamburg, 1997.