SC-CFS: Smartcard Secured Cryptographic File System

Naomaru Itoi

Center for Information Technology Integration
University of Michigan
Ann Arbor

`http://www.citi.umich.edu/projects/smartcard/`

Storing information securely is one of the most important roles expected for computer systems, but it is difficult to achieve with current commodity computers. The computers may yield secrets through physical breach, software bug exploitation, or password guessing attack. Even file systems that provide strong security, such as the cryptographic file system, are not perfect against these attacks. We have developed SC-CFS, a file system that encrypts files and takes advantage of a smartcard for per-file key generation. SC-CFS counters password guessing attack, and minimizes the damage caused by physical attack and bug exploitation. The performance of the system is not yet satisfactory, taking 300 ms for accessing a file.

1 Introduction

Storing information securely has been one of the most important applications of computer systems since their introduction. As information technology is being integrated into society rapidly, secure storage is now demanded more strongly, and by more people, than ever. For example, consider the recent incident in which a laptop computer was stolen from the State Department of the United States in January 2000 [20]. Not only the people who deal with highly classified information, but also ordinary people are threatened by hackers, as they store their private data on computers today, e.g., e-mail, financial information, Internet activity history, and medical history.

For the purpose of this paper, we define *secure storage* as "a storage system that protects the secrecy, authenticity, and integrity of the information it stores".[1] Unfortunately, modern commodity computers cannot provide secure storage because of the three prevalent, but inaccurate, assumptions about computer systems. First, modern commodity computers tend to overlook physical security, and lack physical protection; read and write access to computational and storage devices is typically possible by simply opening the cover of a computer. For example, a hard disk drive is easily removed, giving full access to an adversary. Second, bugs in design and implementation of software are unavoidable [8], and can be exploited to give away secrets. Exploitable bugs are found in all ranges of software, and some of them are so serious that they lead to administrative rights (`root`) compromise [11]. Third, passwords are often the weakest link in security systems. Once passwords are stolen, no matter how securely the system is designed and implemented, it becomes vulnerable to impersonation. Passwords can be stolen from memory, from virtual memory backing store [23], in transit through networks [25], or can be guessed with dictionary attack [17].

An obvious countermeasure to theft of secrets is to encrypt the secrets with an encryption key, and protect the key. Matt Blaze has realized this with a *Cryptographic File System for UNIX (CFS)* [2], which transparently encrypts files in a file system [4]. Although CFS adds significant security to current systems, it still suffers from the the problems introduced above. First, CFS relies on user chosen passwords to provide encryption keys, making dic-

[1] Denning defines the desired properties of a communication channel similarly [6].

[2] Throughout this document, we refer to CFS version 1.3.3 by "CFS".

tionary attack possible. An adversary can obtain ciphertext through physical attack or bug exploitation, and can run an off-line dictionary attack. Second, the number of passwords a user can remember is limited. To lower the burden of the user, CFS uses one key to encrypt all the files in a directory tree, which is not as desirable as using one key for each file. If the key is stolen, physically or through exploitation, the files encrypted under the key are revealed. Therefore, the fewer files are encrypted under a single key, the better.

We attack this problem by storing a randomly generated user key on a smartcard, and generating a file key that is used to encrypt only one file. We have implemented such a system called SC-CFS, based on CFS. Instead of a password, SC-CFS uses the random key on a smartcard to generate file keys, thus thwarting dictionary attack. On host compromise, SC-CFS reveals only the keys of the files that are currently used (and these files are already in memory in the clear, anyway), thus minimizing damage. The design, security considerations, implementation, performance evaluation, related work, and future direction are discussed in this paper.

2 Design

2.1 Cryptographic File System Review

As SC-CFS is based on CFS, it is important to understand how CFS works. CFS consists of a CFS daemon, or *cfsd*, and application programs. `cfsd` is a *Network File System* [21] server daemon (i.e., it provides a file system that can be mounted and be accessed through the NFS protocol) that stores data encrypted. Application programs include `cmkdir`, which creates a CFS protected directory, `cattach`, which prepares a CFS directory for use, and `cdetach`, which reverses `cattach`'s operation.

Readers interested in details of CFS are advised to consult Blaze's paper [4].

2.2 Key Management

The goals of key management are as follows:

- A file key is derived from a master key in a smartcard.

  Only the owner of the smartcard should be able to use the file system. Therefore, a *file key*, which is used to encrypt and decrypt a file, should be derived from the master key in a smartcard. On the other hand, the master key should NOT be derivable from the file key.

- A unique file key is used to encrypt each file.

  A file key is used to encrypt only one file to minimize the damage if it is revealed through host compromise. This property is discussed more in Section 3.

- A file key changes with the associated file.

  When a file is written, its associated file key changes to protect the new file content; this provides forward secrecy. Consider the following scenario: a file key is stolen through host compromise. The file content is revealed to the adversary. Later, the file content is updated by a user. If the file key does not change, the new content is also available to the adversary. To avoid this, the file key should change on every update.

To achieve these goals, we designed the following key management scheme.

- A randomly generated master key is stored in a smartcard.
- `cfsd` uses a file's inode number and a timestamp of last modification as a seed of the file. Each entry is 4 bytes long, so the seed is 8 bytes long ({inode#, timestamp}).
- `cfsd` sends the seed to the smartcard. The card replies with the SHA1 hash result of the seed concatenated with the master key. (SHA1{inode#, timestamp, K_{user}}). This is 20 bytes long.
- `cfsd` further hashes the result into an 56-bit DES key, and uses it to encrypt and decrypt the file.

2.3 Authentication

SC-CFS employs the same authentication mechanism as CFS. A "signature", which is a 4 byte predefined string concatenated with 4 byte random string, encrypted in a way described in the previous key management section, is stored in each CFS directory. When a user starts accessing the directory, `cattach` tries to decrypt the signature. If `cattach` recovers the predefined string correctly, the user has entered the right password (in CFS) or used the right smartcard (in SC-CFS), so he is allowed to enter the directory.

In SC-CFS, before a smartcard is used, the correct *Personal Identification Number (or PIN)* must be typed. The PIN is a 3 - 8 digit number, which protects the information in the smartcard when it is lost or is stolen. The adversary who owns the smartcard cannot use it without knowing the PIN, as the smartcard blocks after some fixed number, say three, wrong PINs are entered.

2.4 Caching

CFS employs partial encryption of a file to minimize the performance overhead introduced by encryption. When a block (8 byte) in a file is updated, it is first XOR'ed with a precomputed string, encrypted with a sub key, and then XOR'ed with another precomputed string. The two precomputed strings and the sub key are pseudorandomly generated, based on the directory key [3]. The advantage of this approach over a chaining mode encryption, such as DES-CBC, is that a file can be partially updated. Chaining mode encryption requires the entire file to be encrypted at once.

As one of our goals is to change a key with every update of the associated file, we do not use this partial encryption approach. Instead, every write re-encrypts the whole file, . Therefore, DES-CBC is used.[4] This introduces potentially prohibitive performance overhead because of paging. In most UNIX systems, a file consists of several 4096 byte pages. A write operation to a long file is split into multiple 4096 byte writes. For example, to write a 1 Mbyte file, 256 write operations are necessary. We cannot afford to change a file key and encrypt the entire file 256 times. To counter this, a single file cache is introduced.

The cache loads a file when it is first accessed, and decrypts it. When the file is closed, it is encrypted under an updated file key and written back to the backing store. Because NFS does not have a `close` operation (NFS server is stateless), writeback happens in one of the following events:

- Another file enters the cache.
- Once a minute.
- CFS directory is detached.

[3] A *directory key* is a key used to encrypt files in a directory. This is entered by a user.

[4] We still could have used partial encryption to achieve partial reads. The decision to use DES-CBC may be reconsidered in the future.

3 Security Consideration

We discuss the security of our approach here, mainly in comparison with CFS. Another cryptographic file system, *Transparent Cryptographic File System (or TCFS)*, has a key management system similar to CFS [5]. Discussion of CFS in this section also applies to TCFS.

3.1 Model

We start with constructing a model of our system. The model consists of the following participants:

Alice (A) A user who uses CFS or SC-CFS.

Host A host computer that runs CFS or SC-CFS.

Smartcard A smartcard that plays the key generation role in SC-CFS.

Backing Store A backing store for CFS or SC-CFS. This may be any file system, e.g., a local file system or a network file system.

Mallory (M) An adversary.

3.2 Threats

We make the following assumptions in our model.

1. Mallory can compromise a host.

   Mallory can exploit security holes of the host, or physically access the host and overwrite the system administrator's password. Mallory can read and modify any information on the host.

2. Mallory cannot substantially change the behavior of the host.

   By Assumption 1, Mallory is able to install a Trojan horse in the host, which, for example, steals decrypted files. However, we assume this attack is impossible because:

   - Maintaining Trojan horses is hard, as Alice can find them by looking at change of file contents and logs.

- It becomes much harder if Alice uses application integrity checker, such as Tripwire [16].
- It becomes even harder if Alice uses hardware based integrity checker, such as AEGIS [1] and sAEGIS [13].

3. Mallory cannot compromise the smartcard.

   Mallory can neither read nor modify any information in a smartcard. She cannot influence the behavior of a smartcard.

4. Cryptographic operations are strong.

   Our principal cipher is DES, which is assumed impossible to compromise in reasonable amount of time. This may not be a good assumption any more in the age of fast DES crackers [7]. DES should be replaced with triple-DES in the future.

   Also, our principal hash function, SHA1, is assumed to be collision free.

3.3 Attack

Key Theft

If a host is compromised (possible by Assumption 1), keys can be stolen in CFS and SC-CFS:

In CFS, the key that encrypts the current working directory is stolen. As a result, all the files in the directory are revealed. Unless the key is explicitly changed, all the files will be accessible by the adversary.

In SC-CFS, the key that encrypts the file currently in the SC-CFS cache is stolen. The rest of the files in the file system are safe. The master key is safe because it is in a smartcard (Assumption 3). When the file is updated, it is encrypted under a different key, so it becomes safe again.

SC-CFS is more secure than CFS because when a key is stolen, only one file can be decrypted by the key. This file is being used by an application, so it resides in the clear in memory, and is revealed on host compromise, anyway. In contrast, when a directory key is stolen in CFS, all the files in the directory tree, including the ones that are not opened, are revealed.

CFS takes this "key per directory tree" approach to avoid forcing a user to remember many passwords. In SC-CFS, a smartcard remembers a randomly generated master key, and generates file keys, eliminating this problem.

Storage Theft

Storage theft is sometimes more easily accomplished than host compromise, thus requires special attention.

In CFS, the keys are derived by user passwords, and are vulnerable to dictionary attack. An adversary who steals a hard disk can run off-line dictionary attack as follows:

- Pick a password.
- Generate a sub key and random strings, as CFS does.
- Apply reversed CFS encryption operation to an encrypted file.
- If this recovers a readable text, this is the right key. If it does not, pick another password and try again.

Many sophisticated password crackers are published (e.g., John the Ripper [22]), and can be used to implement such an attack.

In SC-CFS, the master key is a random number, so it is not vulnerable to dictionary attack. By Assumption 4, brute force attack on the master key is also impossible.

On-Line Attack

In both CFS and SC-CFS, user authentication is performed by `cattach`, with a password in CFS and with a PIN in SC-CFS. As a consequence, if Mallory compromises the host (possible by Assumption 1) while Alice is using CFS or SC-CFS, she is able to impersonate Alice.

This causes more serious damage to CFS than to SC-CFS because with CFS, Alice has no way knowing Mallory is accessing her files. With SC-CFS, one can take advantage of physical isolation of a smartcard to counter this problem. For example, if a LED box that indicates data transmission via a serial port is installed on Alice's computer, she knows when Mallory is accessing her files. Furthermore, a display on the smartcard reader that displays the name of the accessed file and a pad on the reader that asks for a PIN on accessing files are useful.

The problem of on-line attack is a potent threat to almost all smartcard based systems because current smartcards do not have a secure I/O path with users. This is an important problem. A smartcard reader with a built-in PIN pad can solve this problem partially, i.e., it prevents PIN theft. SPYRUS's Rosetta Personal Access Reader 2 is an example of such readers [26].

Virtual Memory Compromise

Niels Provos has pointed out that virtual memory backing store may contain critical secrets even though application programs delete them [23]. By reading a hard disk which is used as the backing store, Mallory is able to recover secrets.

In CFS, the user master key and the directory keys may be in virtual memory. In SC-CFS, the user master key is in a smartcard, so only the file keys are vulnerable.

4 Implementation

Host-side implementation was tested on Linux-2.2.12 and OpenBSD-2.7. NFS is a standard protocol, so this should run on almost any UNIX. Smartcard-side implementation is specific to Schlumberger Cyberflex Access smartcard. Because Cyberflex Access is a Java card, we refer to the smartcard-side program as "SC-CFS applet".

SC-CFS has been implemented as extension to CFS. The implementation is divided into the following parts: modification to `cfsd`, `cattach`, `cmkdir`, and implementation of the SC-CFS applet. Here we discuss each part.

- Modification to `cfsd`

  In CFS, `cfsd` stores {inode#, creation time} in a file called `.pvect_`*encrypted-filename*. SC-CFS uses the same file. [5] First, `cfsd` is modified to store a modification time instead of a creation time, as the modification time is used as a seed of a file key in SC-CFS.

  Then, the single-file cache described in Section 2.4 is implemented. Finally, `read` and `write` operations are modified to access data through the cache.

- Modification to `cattach`

  When `cattach` is invoked with `-p port-number` option, it asks for a PIN instead of a password and then sends it to `cfsd`. `cfsd` initializes the smartcard, sends the PIN to the smartcard, and then carries out the card authentication described in Section 2.3.

- Modification to `cdetach`

  When `cdetach` is invoked, `cfsd` cleans up the cache and terminates the connection with the smartcard.

- Modification to `cmkdir`

  When `cmkdir` is invoked with `-S` option, it creates a signature described in Section 2.3 in the newly created SC-CFS directory.

- Implementation of SC-CFS applet

  The master key is stored in a file in a smartcard called `''ke''`, or `0x6b65`. This file is configured so that it cannot be accessed without going through the applet. The applet reads this file only after the correct PIN is presented. Key generation is simple: the applet concatenates the 8 byte seed to the 16 byte master key, hashes it with SHA1, and returns the result to `cfsd`.

[5] CFS does this instead of using information in the vnode structure, as the information changes on undesirable occasions, e.g., when a file is backed up and is resumed, or its modification time is changed by `touch`.

5 Performance Evaluation

We have evaluated the performance of SC-CFS in comparison with CFS and a local file system (EXT2). First, the result of the Andrew Benchmark Test [10] is reported to show user response time. Then, we look into the details of SC-CFS's most expensive operation: smartcard access.

The result shows that SC-CFS is significantly slower than CFS when it accesses a smartcard to generates keys. Most of this penalty is due to the slow speed of a smartcard.

All the measurements have been carried out on Linux-2.2 with 400 MHz AMD K6 and on Cyberflex Access smartcard. All the numbers reported are in seconds, and are average of 5 trials.

5.1 Round Trip Time

The Andrew Benchmark (ABM), a standard file system benchmark test, is used to measure the overhead of SC-CFS. ABM has five phases: MakeDir (`mkdir`), Copy (`cp`), ScanDir (`ls -l`), ReadAll (`grep`), and Make (`cc`). Source code of C programs used in the Make phase is slightly modified from the original Andrew Benchmark to make the test runnable on Linux-2.2 [6]. The results are shown below. The numbers are in seconds.

	Local	CFS (sec)	SC-CFS (sec)
MakeDir	0	0.2	0.2
Copy	0.6	1.0	21.8
ScanDir	1.2	1.6	1.0
ReadAll	2.0	3.0	22.6
Make	5.0	7.8	29.6

SC-CFS works as efficiently as Local and CFS when it does not need to access a smartcard (MakeDir and ScanDir [7]). However, in the other cases (Copy, ReadAll, Make), SC-CFS is much slower.

This performance impact is clearly due to the slow speed of a smartcard. Key generation, the only service the smartcard provides, takes 0.31 second. The following table shows: (1) the number of accesses to a smartcard, (2) (1) $\times$ the average smartcard access time (0.31), and (3) the difference between the round trip time of SC-CFS and CFS. The second column and the third are very close, showing that the most of the performance overhead is for smartcard.

	#acc	#acc $\times$0.31(s)	SC-CFS−CFS(s)
MakeDir	0	0	0
Copy	70	21.7	20.8
ScanDir	0	0	-0.6
ReadAll	70	21.7	19.6
Make	75	23.3	21.8

5.2 Detailed Look

As smartcard access is seen to be the bottleneck of SC-CFS, this part deserves special attention. Detailed performance evaluation was carried out on Cyberflex Access, which communicates at 57.6 Kbps with the host.

SC-CFS's smartcard operation involves two APDUs [8]: One is `generate_key`, which sends an 8 byte seed to the smartcard and invokes the key generation method inside the smartcard. The other is `get_response`, which asks the smartcard to return the result of key generation. A smartcard standard ISO 7816-4 [12] defines the `T=0` communication protocol, which Cyberflex Access adopts, to be unidirectional, i.e., a smartcard can either send or receive data in one APDU. Therefore, in addition to `generate_key` APDU, `get_response` APDU is necessary. These two APDUs are sent to the smartcard consecutively.

The following table shows the breakdown of the two APDUs. "`generate_key` APDU overhead" is time spent for sending a seed to smartcard, invoking the method, and preparing a buffer for returned data. Because this cannot be broken down further, it is shown as one operation.

operation	time (sec)
Hash (SHA1) 24 byte into 20 byte	0.15
`generate_key` APDU overhead	0.10
Select root in file system	0.01
Select key file "ke" in file system	0.01
Read 16 byte from key file	0.01
`get_response` APDU (20 byte)	0.01
total	0.29

The cost boils down to two dominating operations: SHA1 hash function and `generate_key` APDU overhead. These two are necessary operations, and we cannot improve the performance of them without modifying the smartcard. This points out the necessity of smartcards that execute cryptographic operations faster, with lighter method invocation overhead.

6 Related Work

There exist several remotely keyed encryption algorithms. The remotely keyed encryption is a way to encrypt bulk data with a key in a smartcard, where only small amount of work is done on the smartcard, and the rest is done on a fast host. Our session key generation scheme is one of them. Other examples include RKEP by Matt Blaze [3], another one by Blaze [2], ReMaRK [19] and ARK [18] by Lucks, and one by Jakobsson et al. [15].

We have chosen our keying scheme because this is implemented the fastest on the smartcard we used (Cyberflex Access). It is implemented with one

[6] We added five global variables, removed two `getchar()`s, and changed options to `ar`. None of them should alter performance significantly.

[7] File attributes retrieved by `stat()` are not encrypted.

[8] An *APDU* is a command sent to a smartcard from a host. Readers interested in smartcard concepts are advised to consult a reference text [9].

smartcard call, in which a hash function is called once. RKEP requires one smartcard call with two encryption calls and one hash function call. The other Blaze requires one smartcard call with one encryption call (encryption is more expensive than hash function in Cyberflex Access). ReMaRK requires two smartcard calls with two encryption function calls and one hash function call. ARK requires two smartcard calls with two random permutation calls and two random function calls. Jakobsson requires one smartcard call with one encryption call.

Our keying scheme appears to protect a master key and generates good session keys. However, not being cryptographers, we could not prove our scheme to be as secure as the other schemes in this paper. Whether our scheme is best for SC-CFS or we should choose one of the the other schemes is under discussion.

7 Future Direction

7.1 Administration Tools

With the current SC-CFS prototype, a user has to manually update his master key and PIN via our smartcard communication tool called `pay` [24]. Automated tools to do this should be provided.

7.2 Performance Improvement

Clearly, performance overhead is a large obstacle against wide deployment of SC-CFS. 300 millisecond overhead per file is acceptable for some applications, for example, word processing, but is not for others, such as scanning a large number of e-mail messages for a string, or a query operation on a large database. Therefore, performance improvement is essential.

Unfortunately, as shown in Section 5.2, the overhead is dominated by individual operations in a smartcard, which we, as application developers, cannot change. We hope new smartcards or other similar devices will achieve much higher performance in the near future.

To improve the performance of SC-CFS with current smartcard technology, it is possible to compromise between "key per directory tree" approach (CFS) and "key per file" approach (SC-CFS). The former is more efficient, but the latter is more secure. Depending on the security and performance requirements of an environment, middle ground implementation may be useful, e.g., "key per n files" approach, or caching file keys.

8 Conclusion

We have developed SC-CFS, which improves the security of CFS by integrating a smartcard as a personal secure storage of a key.

The following three aspects highlight the value of this work.

Improvement to important software.

As introduced in Section 1, the increasing threat of physical attack demands a way to protect secrets in a computer. CFS (and all the other file systems that protect files through encryption) is a secure and seamless solution to this problem. This work improves CFS in two important properties: security and convenience. SC-CFS is more secure than CFS because (1) the master key is a random number instead of a password, (2) the user master key is not exposed to the host, and (3) a stolen file key can reveal only one file. It is also more convenient than CFS because all a user has to remember is a short PIN, rather than multiple long passwords.

Important application for smartcards.

We believe that widespread deployment of secure hardware is essential to the security of computer systems. Systems are as secure as the weakest link, and reliance on user passwords is often the weakest link. Granted, passwords can be made stronger by choosing good ones and by changing them often. However, widespread deployment of security-critical Internet services requires a human to maintain many passwords: computer login, file system authentication, newspaper homepages, e-commerce homepages, online banks, web portals, and so on. Realistically speaking, it is impossible for a human to maintain so many good (therefore hard to remember) passwords. She will end up using the same password for many services, or will write the passwords down somewhere. Besides, she does not want to type passwords all the time. Smartcards solve this problem nicely by securely storing keys. Therefore, we wish to contribute to the widespread deployment of smartcards, and this work is an important step toward the goal, as secure storage is an important and suitable application of a smartcard (authentication being another [14]).

Remark on smartcard performance

Performance evaluation in Section 5 shows how important a fast smartcard is. In recent years, smartcards have matured in terms of functionality and reliability. However, we have not seen significant performance improvement, even though microprocessors have sped up by 5 to 10 times.

9 Availability

SC-CFS has been tested on Linux 2.2 and OpenBSD 2.7. The source code of SC-CFS is available at CITI homepage:

`http://www.citi.umich.edu/projects/smartcard/sc-cfs.html`

Acknowledgment

We thank Niels Provos for suggesting a high speed key generation method. Peter Honeyman and Jim Rees have advised us through this project, and suggested a file caching idea.

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

References

[1] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *1997 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.

[2] M. Blaze. Key management in an encrypting file system. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 27–35, Boston, MA, USA, 6–10 1994.

[3] M. Blaze. High-bandwidth encryption with low-bandwdith smartcards, 1996.

[4] Matt Blaze. A cryptographic file system for UNIX. In *Proceedings of 1st ACM Conference on Computer and Communications Security*, pages 9–16, Fairfax, Virginia, November 1993. ftp://ftp.research.att.com/ dist/mab/cfs.ps.

[5] G. Cattaneo, G. Persiano, A. Del Sorbo, A. Celentano, A. Cozzolino, E. Mauriello, and R. Pisapia. Design and implementation of a transparent cryptographic file system for UNIX. Unpublished Technical Report. Dip. Informatica ed Appl, Universita di Salerno. Available via ftp in ftp://edugw.dia.unisa.it/pub/tcfs/docs/tcfs.ps.gz., July 1997.

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

[7] Electronic Frontier Foundation. *Cracking DES - Secrets of Encryption Research, Wiretap Politics & Chip Design.* O'Reilly & Associates, Inc., 1 edition, 1998.

[8] Jr. Frederick P. Brooks. *The Mythical Man-Month : Essays on Software Engineering.* Addison-Wesley, 2 edition, July 1995.

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

[10] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51 – 81, Feb. 1988.

[11] John D. Howard. Security incidents on the internet. In *Proceedings of INET 98.* Internet Society, 1998. http://www.comms.uab.es/inet99/inet98/ 2d/2d_3.htm.

[12] The International Organization for Standardization and The International Electrotechnical Commission. *ISO/IEC 7816-4 : Information technology - Identification cards - Integrated circuit(s) cards with contacts*, 9 1995.

[13] Naomaru Itoi, William A Arbaugh, Samuela J Pollack, and Daniel M Reeves. Personal secure booting. Technical report, Center for Information Technology Integration, 2000. To appear in ACISP 2001, Australia. Technical Report at http://www.citi.umich.edu/ techreports/.

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

[15] M. Jakobsson, J. Stern, and M. Yung. Encrypt small, 1999.

[16] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: A file system integrity checker. Technical report, Purdue University, 1995. CSD-TR-93-071.

[17] Daniel V. Klein. Foiling the cracker: A survey of, and improvements to, password security. In *UNIX Security Workshop II*, pages 5 – 14. USENIX Association, 1990.

[18] S. Lucks. Accelerated remotely keyed encryption. *Lecture Notes in Computer Science*, 1636:112–123, 1999.

[19] Stefan Lucks. On the security of remotely keyed encryption. In *Fast Software Encryption*, pages 219–229, 1997.

[20] Christopher Marquis. Deep security flaws seen at state dept, May 2000. http://www.fas.org/sgp/news/2000/05/ nyt051100.html.

[21] Sun Microsystems. Network filesystem specification. Network Working Group, Request For Comments 1094, March 1989.

[22] Open Wall Project. John the ripper. http://www.openwall.com/john/.

[23] Niels Provos. Encrypting virtual memory. In *Proceedings of 9th USENIX Security Symposium*, August 2000.

[24] Jim Rees. Iso 7816 library, 1997. http://www.citi.umich.edu / projects / sinciti / smartcard / sc7816.html.

[25] Dug Song. dsniff. http://www.monkey.org/dugsong/dsniff/.

[26] Spyrus. http:// www.spyrus.com/.