Integrating Secure Hardware into Modern Security Systems: Authentication, Secure Storage, and Secure Bootstrap

by

Naomaru Itoi

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Computer Science & Engineering) in The University of Michigan 2001

Doctoral Committee:

Professor Peter Honeyman, Chair
Professor Farnam Jahanian
Professor Atul Prakash
Professor Paul Resnick
Professor Sean Smith, Dartmouth College
Acknowledgment

First and foremost, I am grateful to Schlumberger, who has supported my research through the entire Ph.D. program.

I would like to express my gratitude to my research advisor, Professor Peter Honeyman, for his wise and practical guidance. He has taught me not only knowledge, but also aggressiveness and sincerity.

I am deeply grateful to Professor Farnam Jahanian, Professor Atul Prakash, Professor Paul Resnick, and Professor Sean Smith, for kindly serving on my doctoral committee.

None of my work could have been done without the assistance of staff and students at CITI. In particular, I appreciate Jim Rees for his seemingly unlimited knowledge on smartcards and operating systems. Also, help from Evan Cordes, Tomoko Fukuzawa, and Niels Provos was invaluable.

I would like to acknowledge several people outside the university. Timothy M. Jurgensen, Mike Montgomery, Nevile Patterson at Schlumberger, and Andrew Webb at Giesecke & Devrient America, Inc. provided me with smartcards and the smarts to use them effectively. Elaine Palmer, Professor Sean Smith, Ronald Perez, and Dawn Song at IBM Research provided me with secure coprocessors and valuable discussion. Professor William Arbaugh has provided me with an AEGIS integrated personal computer, tips to use it, and a lot of good advice. David Jablon has kindly allowed me to use his patented protocol, SPEKE, for non-commercial use. I have also received great help from freeware developers from all over the Internet, namely, developers of Kerberos, PAM, Arla, GRUB, FIFS, and CFS.

I thank friends of mine, Tomoko Fukuzawa, Yuko Imamura, Yosuke Ishikawa, Olga Kornievskaya, Li Li, Asako Maruyama, Takahiko Masuda, Tomoe Matsuoka,
Kazuyoshi Nakamichi, Akira Nagamori, Gi-Joon Nam, Mana Narita, Yoshiaki Ohkubo, David Ong, Hirotaka Saito, Mitsuyo Sakamoto, Mitsuhiro Takahari, Haruna Takahashi, Taketoshi Ushiama, Jean Wu, Joon-Hwan Yi, the Indonesian soccer team, the Thai soccer team, and the Japanese football team (the Chipmunks) for assisting me, encouraging me, playing with me, and eating with me. I could not achieve this without them.

Finally, I appreciate my parents and three sisters for letting me play with computer toys for such a long time.
Contents

Acknowledgment ii

List of Figures viii

List of Tables x

Chapter

1 Introduction 1

1.1 Introduction .................................................. 1

1.2 Vulnerabilities of Modern Computer Systems ............... 4

1.2.1 Untrustworthy Computer Systems .......................... 4

1.2.2 Reliance On Passwords ...................................... 6

1.3 Threat Model .................................................. 7

1.4 Solution: Integration of Secure Hardware with Current Security Systems 9

1.5 Limitation: Vulnerability to On-Line Attack ............... 10

1.6 Dissertation Composition ..................................... 10

2 Related Work 12

2.1 Untrustworthy Computer Systems ............................... 12

2.2 Reliance On Password .......................................... 13

2.3 Secure Hardware ............................................... 15
2.3.1 Smartcards .......................... 16
2.3.2 Secure Coprocessors ................. 18
2.4 Kerberos ............................... 20
2.5 Cryptographic File System ............ 23
2.6 AEGIS: Secure Bootstrap Process .... 24
2.7 Research Aiming at Similar Goals .... 26
   2.7.1 Type-Safe Languages ............... 26
   2.7.2 Secret Sharing .................... 26
   2.7.3 Sandboxing ....................... 27
   2.7.4 Top-Down Secure Operating System 27

3 Secure Hardware Integration with Kerberos 29
   3.1 Smartcard Integration with Kerberos V5 Client 29
      3.1.1 Introduction ..................... 29
      3.1.2 Design .......................... 30
      3.1.3 Security Discussion ............. 33
      3.1.4 Implementation .................. 37
      3.1.5 Performance Evaluation ......... 41
      3.1.6 Discussion ...................... 44
      3.1.7 Section Summary ................ 46
   3.2 Secure Coprocessor Integration with Kerberos V5 Server 48
      3.2.1 Introduction ..................... 48
      3.2.2 Design .......................... 49
      3.2.3 Security Discussion ............. 52
      3.2.4 Implementation .................. 56
      3.2.5 Performance Evaluation ......... 60
3.2.6 Discussion ......................................................... 64
3.2.7 Section Summary .............................................. 69
3.3 Internet Extension to Smartcard Kerberos Client ............ 70
  3.3.1 Introduction .................................................. 70
  3.3.2 Design ........................................................ 71
  3.3.3 Security Discussion ......................................... 77
  3.3.4 Implementation .............................................. 79
  3.3.5 Performance Evaluation ................................. 86
  3.3.6 Section Summary ........................................... 90

4 Secure Hardware Integration with Secure Storage 92
  4.1 Smartcard Filesystem .......................................... 93
    4.1.1 Introduction ............................................... 93
    4.1.2 Design .................................................... 95
    4.1.3 Implementation ........................................... 98
    4.1.4 Performance Evaluation ................................. 104
    4.1.5 Discussion ............................................... 107
    4.1.6 Section Summary ....................................... 110
  4.2 Smartcard Secured Cryptographic File System ............... 113
    4.2.1 Introduction ............................................... 113
    4.2.2 Design .................................................... 114
    4.2.3 Security Consideration ................................... 117
    4.2.4 Implementation ........................................... 120
    4.2.5 Performance Evaluation ................................. 122
    4.2.6 Discussion ............................................... 125
    4.2.7 Section Summary ....................................... 126
5 Personal Secure Booting with Smartcards 128

5.1 Personal Secure Booting ........................................... 129
  5.1.1 Introduction .................................................. 129
  5.1.2 Design ....................................................... 129
  5.1.3 Security Consideration ................................. 134
  5.1.4 Implementation .......................................... 139
  5.1.5 Performance Evaluation .............................. 141
  5.1.6 Discussion ............................................... 142
  5.1.7 Section Summary .................................. 143

6 Conclusion 145

  6.1 Summary ..................................................... 145
  6.2 Contribution ............................................... 146
  6.3 Future Direction ........................................ 148

Appendix A 102

  Published Papers ............................................. 102

Bibliography 108
List of Figures

2.1 Kerberos Authentication Service Protocol .................................. 21

3.1 Kerberos Authentication Service and Ticket Granting Service without Smartcard ................................................................. 31

3.2 Kerberos Authentication Service and Ticket Granting Service with Smartcard ................................................................. 32

3.3 DES-CBC-MD5-Smartcard Cryptosystem Entry ................................. 38

3.4 Authentication Service without the 4758 ..................................... 51

3.5 Authentication Service with the 4758 ......................................... 51

3.6 Ticket Granting Service without the 4758 ................................... 52

3.7 Ticket Granting Service with the 4758 ..................................... 52

3.8 Kerberos / 4758 Performance - Total Time, 4758 Time, and Communication Overhead ........................................................... 62

3.9 SPEKE protocol ........................................................................ 75

3.10 SPEKE implemented with smartcard ........................................ 77

3.11 Implementation Overview .......................................................... 80

3.12 SPEKE Data Structure .............................................................. 81

3.13 EKE Protocol ........................................................................... 88

4.1 SCFS Design ............................................................................ 97

4.2 Reading 8 Byte Data from Smartcard .......................................... 99
4.3  Total Time, Smartcard Time, and SCFS Overhead ................................ 104
4.4  SCFS Read and Write Performance ............................................... 106
4.5  Application that uses SCFS .......................................................... 111
4.6  Application that uses OCF or PC/SC ............................................. 111

5.1  Boot Strap Process ........................................................................ 133
5.2  Smartcard - Workstation Communication Protocol .......................... 135
## List of Tables

2.1 Comparison between Smartcard and Secure Coprocessor ........... 20

3.1 Authentication Service with Smartcard Time Line ................. 42

3.2 Authentication Service with Smartcard Time Breakdown ........... 42

3.3 Ticket Granting Service with Smartcard Time Line ................. 43

3.4 Ticket Granting Service with Smartcard Time Breakdown ........... 44

3.5 Kerberos / 4758 Notation ........................................ 51

3.6 Kerberos / 4758 - Total Time, 4758 Time, Communication Overhead 61

3.7 Kerberos / 4758 Performance - Authentication Service Breakdown .. 63

3.8 Kerberos / 4758 Performance - Ticket Granting Service Breakdown . 63

3.9 Kerberos on SPEKE Timeline ....................................... 87

3.10 Kerberos on SPEKE Breakdown .................................... 87

3.11 EKE Timeline ......................................................... 89

3.12 EKE Breakdown ..................................................... 90

4.1 SCFS Overhead for Read and Write ................................. 105

4.2 SCFS Overhead for Other Operations ................................. 105

4.3 SCFS Read and Write Performance .................................. 106

4.4 SC-CFS Andrew Benchmark Result ................................ 123

4.5 SC-CFS Smartcard Performance Impact ............................. 123

4.6 SC-CFS Smartcard Performance Breakdown .......................... 124
Chapter 1

Introduction

This chapter introduces the problems I address, the approach I take, and the environment I work with in this dissertation.

1.1 Introduction

With the rapid integration of information technology into society, the demand for computer system security is soaring. Despite decades of extensive research on information security, most computer systems are for the most part vulnerable to common forms of attack: physical breach, software bug exploitation, and password compromise. This trend reflects the prevalent, but inaccurate, assumptions about computer systems and the quality of user passwords. Although computer security depends on, amongst other things, trusted computing, authentication, cryptography, access control, secure communication, and formal analysis, my focus is on trusted computing and authentication because of the critical roles they play in computer security. For the purpose of this dissertation, I define a trusted computer as “a computer system that behaves as its users intend, without damaging the users’ computing resources or damaging / leaking their information”. ¹

¹This definition is narrower than the one used in the US Trusted Computer Security Evaluation Criteria [78], in which the word “trusted” includes access control, covert channel analysis, etc. My
A major problem today is that modern commodity computers are not trustworthy because (1) they tend to overlook physical security, and (2) they are inevitably vulnerable to the exploitation of software bugs. An equally important problem is that reliance on user chosen passwords is often the weakest link in a security system. Users tend to choose passwords that are easily remembered, and such passwords often depend on names or other terms of significance to the user, which in turn quickly narrows the ranges of possible passwords and makes them easier to guess. Moreover, an adversary can steal passwords by compromising a computer that processes passwords. This dissertation explores an approach to security that counters these problems by integrating secure hardware into existing security infrastructures.

Conventionally, the problem of trusted computing has been tackled by such approaches as access control mechanisms [23], layered architecture [76], sandboxing [35, 101], application level integrity check [54]. However, all of these approaches trust the underlying hardware and operating system kernels, and are of little use if the components are compromised. Furthermore, some of them require new operating systems, which compromises ease of deployment.

As for the issue of password protection, most research has focused on either password policies that make passwords harder to guess, or protection of the password information necessary to run password guessing attacks. The former approach is quite effective, but forces users to remember and use complicated passwords. Furthermore, it does not completely defeat the attack: the more sophisticated the password policies, the more sophisticated the password crackers. The latter approach is limited, too, because of the exposure to physical attack and bug exploitation.

These problems, however, can be solved by secure hardware, which provides a

\[\text{definition is closer to the ones used by Brewer et al. [10], Goldberg et al. [35], and Loscocco et al. [69]}\]
physically secure, relatively bug free storage and computational device. Yet, currently, secure hardware products have not been widely used because they are not convenient enough. I take an experimental approach to address this problem; I integrate existing secure hardware into existing security infrastructures, with minimum modification to the infrastructures in terms of coding, interfaces, and performance. The thesis of this dissertation is, therefore:

The security of existing computer security infrastructures can be significantly improved by integrating existing secure hardware. This approach is feasible in the real world, is convenient for users, and is cost effective for developers and system administrators.

I prove this thesis by integrating commodity secure hardware into three important security applications: authentication, secure storage, secure bootstrap, as well as by evaluating their security benefits and performance. The amount of modification made to the original systems is not significant: usually 1 - 2 months' work per application by a graduate student (me). The resulting systems have much stronger security, and are often more convenient than the original systems. They are deployable without much introduction cost except the cost of additional hardware. They are slower than the original systems; the introduced overhead is not significant in some of the projects, but it is in the others. In such projects, I optimize the software heavily, and report where the bottlenecks are.

In the rest of this chapter, I explore the problem space, and state the threat model and the solution generally. This dissertation consists of several projects, and the problems, the threat models, the solutions are in some ways peculiar to each project. Therefore, detailed description of these are given in each project’s section.
1.2 Vulnerabilities of Modern Computer Systems

In this section, I elaborate the two very common vulnerabilities of modern computer systems, namely, untrustworthy computer systems, and reliance on passwords.

1.2.1 Untrustworthy Computer Systems

The first problem is the lack of any trustworthy foundation in modern computer systems. An untrustworthy foundation makes the security system on top of it a castle in the air; it can be cheated by the underlying components in several ways. A modern computer system is highly abstracted, and it hides underlying details from a user. Although this is convenient for the user, she does not know if the computer is doing what she told it to. For example, she may be cheated by application programs. Consider a Kerberos client that steals the user’s password [6]; an SSL client that leaks plain text packets [10, 69]; Tripwire that only pretends to check files [41]; or a web browser that shows a certificate but does not check its validity. Such systems provide the illusion of strong security, but in reality, they are seriously flawed. Worse yet, these threats are unavoidable because (1) physical security is overlooked in commodity hardware, and (2) software bugs inevitably introduce security threats.

The fear of physical threat is growing; it is highlighted by recent incidents. A laptop computer disappeared from a conference room in the State Department of the United States in January 2000 [19]. Two hard disks disappeared from the U.S. nuclear laboratory at Los Alamos in June 2000. Both of the lost items contained highly classified military information. Other such incidents include a laptop computer stolen from an employee of Britain’s domestic security service in March 2000 [93], and a medical server stolen from a university hospital in the U.S. [37].

There are several reasons why modern computers are vulnerable to physical at-
tacks. First, a modern computer environment consists of distributed personal computers (or PCs) and workstations connected by networks, unlike mainframe computers of the past in isolated computer centers. Such an environment is difficult to protect because computers are geographically distributed, making it difficult to restrict unauthorized physical access. Widespread use of laptop computers makes it even more difficult. Second, PCs and workstations have weak 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, and gives away full access to an adversary. Some computers have locks, but the locks tend to be of low quality and are easily defeated [29]. Third, operating systems assume physical security. For example, many UNIX systems have an emergency rescue mode called the single-user mode, in which the computer grants full access to any resource to the person who boots it. A user or an adversary can enter this mode by selecting an option when the operating system boots up, in some cases, without even typing the system administrator’s password.

Even if we solve these hardware problems, computers are still vulnerable because of software problems. Bugs in design and implementation of software are unavoidable[32], and can be exploited to cause serious damage. Exploitable bugs are found in all ranges of software: from operating systems (e.g., erroneous permission of DLL cache on Windows NT 4.0 [26]), daemons (e.g., buffer overflow in wu-ftpd [17]), to applications (e.g., buffer overflow in sendmail [16]). Some of them can be so serious that they lead to administrative rights (root) compromise, allowing an adversary to run arbitrary code with administrative privileges. Once an adversary obtains administrative privileges, he can access any resource in the computer, install a Trojan horse, and can even modify the operating system kernel to hide the Trojan horse [41]. Many such flaws are reported to the computer security community,
which publishes countermeasures as soon as vulnerabilities are found (e.g., bugtraq [13] and CERT-CC [15]). However, searching for vulnerabilities is an endless task: to debug any software completely is a Herculean task; to do so for the thousands of new programs coming out every year is impossible.

1.2.2 Reliance On Passwords

The second problem is that passwords are often the weakest link in security systems. The importance of protecting passwords cannot be overstated - once passwords are lost, the entire security system becomes vulnerable to impersonation. Passwords are used quite widely; almost all user authentication systems rely on passwords, e.g., workstation login, e-mail, Kerberos, and SSH. Unfortunately, as widely used as passwords are, a secure solution they are not.

First, they can be stolen. As described above, a modern computer can be compromised. A computer compromise reveals passwords stored in it, e.g., in an application’s process memory, or in the virtual memory backing store [83]. They can also be stolen by a Trojan horse that forges a login screen and steals a password. They may also be stolen when they travel through the Internet. Clear text passwords are used in some applications, e.g., ftp and telnet, even though they can be replaced with protocols that use encrypted passwords, such as SSH.

Second, they can be guessed. Passwords chosen by users are often easily memorable, and such passwords are easily guessed; they are highly likely to be revealed by password cracking tools, such as John the Ripper [82]. For example, passwords are used in Kerberos, one of the most widely used network authentication mechanisms. In 1995, the University of Michigan’s Kerberos system was experimentally attacked by staff at the Center for Information Technology Integration. The staff used password guessing attack (or dictionary attack), and they quickly obtained 25% of 20,000 tested
passwords, discovering the most common password to be “love” (go figure).

Concerned with this result, the university started enforcing a policy on password choice, but this did not solve the problem completely. The policy was quite strict; a password must be at least 8 characters, it must include a mix of alphanumeric and punctuation characters, it must not be in a dictionary of any language, etc. The policy drastically improved the situation, but password guessing attack is still effective. Five years after the policy came into effect, in 2000, CITI staff were able to obtain more than 2,000 passwords. The most common password found this time was “asdfkl;”. Users always seem to find a way to outsmart a password checker and come up with a trivial password that is not prohibited by the checker (and easily guessed by an attacker).

And who can blame the users for choosing trivial passwords? They are asked for a password in too many occasions, as the number of Internet services that require user authentication explodes: workstation login, remote login, file transfer, e-commerce web sites, mailing lists, banks, web portals, etc. It is impossible to select, remember, and use a different, good password on every system. So users often end up choosing a trivial password, using the same one across different systems, not changing it for a long period of time, writing it down, etc. After all, convenience is what users want. Therefore, to address the problem, passwords must disappear, or at least should be used much less frequently.

1.3 Threat Model

Here I define the environment in which I work. The following assumptions are made throughout this dissertation.

- A workstation can be compromised.
I use the term “workstation” to refer to PCs and other similar kinds of workstations. As Section 1.2.1 describes, a workstation can be compromised in several ways. Therefore, I assume that an adversary can read, write, and destroy any information in a workstation.

- A network can be compromised.

Likewise, I assume an adversary can read, write, and destroy any information on networks that connect workstations.

- Users choose weak passwords.

From the discussion in Section 1.2.2, I assume users choose weak passwords, and an adversary can guess them.

- Secure hardware cannot be compromised.

I use a term secure hardware to refer to computers that are designed to be secure, e.g., smartcards and secure coprocessors. I assume them secure; an adversary cannot read, write, or destroy any information in them, and cannot influence their behavior. I also assume that, with an appropriate user authentication, e.g., with a personal identification number, secure hardware does not leak any information when it is stolen. While secure hardware suffers from newly developed attacks [57, 61], I ignore such attacks in this dissertation because (1) secure hardware is still much harder to compromise than workstations, and (2) secure hardware developers are developing countermeasures to the new attacks.

- Cryptographic operations are strong.

I assume that the adversary has ordinary computational power, e.g., he can neither solve a NP-complete problem nor factor a 1024 bit number. Hence, he
cannot break conventional cryptographic operations, e.g., RSA, DES\textsuperscript{2}, SHA1, and MD5.

1.4 Solution: Integration of Secure Hardware with Current Security Systems

The approach I take in this dissertation is to integrate secure hardware into existing security systems to enhance their security, while maintaining the same interfaces. The following are my design policies.

- Identify the critical secrets that should be protected in the existing systems. For example, keys used for encryption typically should be protected.

- Store the critical secrets in secure hardware.

- Keep the secrets in secure hardware as much as possible. To achieve this, the secrets should be used only in secure hardware. For example, execute all the cryptographic operations that are associated with the keys in secure hardware.

- Replace passwords with randomly generated keys.

- Make sure of the presence of secure hardware when the identity of the user must be proved.

- Take advantage of hardware seeded random number generation when possible.

- Minimize the modification to the existing system for the ease of deployment.

The systems developed with these policies protect critical secrets even if the workstations are compromised, and solve the weakness of passwords.

\textsuperscript{2}This may not be a good assumption any more in the age of fast DES crackers \cite{30}. Nonetheless, I still assume DES is strong because it is relatively simple to replace DES with a stronger algorithm.
As secure hardware being slower than workstations, my approach introduces trade-offs between security and performance. For example, if an entire security system runs on secure hardware, it becomes slow. On the other hand, if it runs entirely on a workstation, it suffers from the vulnerabilities presented above. The challenge of my work is to strike the balance between the security and performance. To answer this question, all of my projects have detailed security discussion and performance evaluation. They should serve as a guideline for developing similar systems.

1.5 Limitation: Vulnerability to On-Line Attack

My approach cannot perfectly defend against on-line attacks. An on-line attack is an attack mounted by an adversary or the Trojan horse installed by the adversary while secure hardware is being used. Because existing secure hardware typically does not have a trusted I/O path with a user, it cannot distinguish the requests from an adversary with the ones from a user. The adversary can take advantage of the service provided by secure hardware.

It is difficult to mount an on-line attack because it requires exact timing and sophisticated techniques to hide the attacker from users. However, it remains as a potent threat until secure hardware will implement a trusted I/O path. In this dissertation, I minimize the damage caused by an on-line attack as much as existing secure hardware allows. Details of the on-line attacks against my systems, and the countermeasures, are described in each section.

1.6 Dissertation Composition

To motivate my work and to underpin my reasoning, the most closely related work is reviewed in Chapter 2. Then I present the implementation of my thesis on three
security systems: authentication, secure storage, and secure booting, in Chapters 3, 4, 5, respectively. I concentrate on authentication because it is a necessary and important step in almost any security system, and secure storage and secure booting because they are demanded strongly. Finally, I conclude this dissertation in Chapter 6.
Chapter 2

Related Work

This chapter reviews the work most closely related to my research. The first two sections motivate my work by reviewing the two problems I address. Section 2.3 introduces secure hardware. Sections 2.4 - 2.6 introduce three security systems into which I integrate secure hardware, namely, Kerberos, Cryptographic File System, and AEGIS Bootstrap Process. In addition, Section 2.7 discusses some security projects that address similar goals with mine. Although they do not directly influence the design of my systems, reviewing them is helpful to grasp where my work stands in the computer security research.

2.1 Untrustworthy Computer Systems

The risk of untrustworthy computer systems have been identified by many researchers. One of the researchers is Bellovin and Merritt, who discuss an attack to Kerberos clients where workstations are untrustworthy [6]. “In a workstation environment, it is quite simple for an intruder to replace the login command with a version that records users’ passwords ...” This attack can obviously be applied to any password-based authentication system.

Another research group, Brewer et al., implemented an attack on the Netscape
browser to illustrate this problem [10]. This attack modifies the browser’s executable code as it travels through NFS. The modified browser uses a fixed key for all SSL connections, enabling an adversary to decrypt all the data transmitted on the connection. They argue that the need of trusted code at end points is overlooked. “Many security products (Kerberos, AFS, NFS, Netscape, Microsoft) have ignored (for all practical purposes) the issue of trusting your executables.”

Other researchers, not only point out this problem, but also propose solutions.

Arbaugh et al. discuss this problem: “... any system is only as secure as the foundation upon which it is built ... We find it surprising, given the great attention paid to operating system security over the years, that so little attention has been paid to underpinnings required for secure operation” [3, 2]. To counter this problem, they developed a secure bootstrap process, AEGIS. AEGIS ensures that every program that runs in the bootstrap process is verified by its underlying layer with a code signing technique. When AEGIS boots an operating system, it is certain that the operating system has not been modified by an adversary. Secure booting is an important piece of a trusted system, but not the only one, as it does not protect a system after booting.

Yee et al. and Checkley address this problem, too. I discuss their work in Section 2.3.

2.2 Reliance On Password

It has long been known that passwords are vulnerable to dictionary attack. The attack was used by Morris and Thompson to attack early UNIX systems [75] in the late 1970s. They applied a one-way hash function to potential passwords in a dictionary, and checked if the result matched with entries in /etc/passwd. “The dictionary search alone, which required only five minutes to run, produced about one
third of the passwords”.

This problem has been addressed with two approaches: making passwords harder to guess, and protecting password information that is necessary to run password guessing attack. Both are effective, but not complete.

The first approach is to force users to follow a password policy to make passwords hard to guess (e.g., a password must be longer than 8 characters, must contain non-alphabetical symbols, etc.) This approach has significantly improved the security of passwords, but not completely. The password policies have become more sophisticated since the research of Morris and Thompson, but so have the password crackers. Dictionary attack was still effective in 1990, when it was used by Klein to reveal 21% of passwords in one week of CPU time [56], and in 2000, when John the Ripper, a dictionary attack based password cracker, revealed 5% of passwords in a Kerberos realm in the University of Michigan. As complex passwords cannot be remembered by users, I do not expect dictionary attack can be defeated by this approach completely.

The second approach is to prevent an adversary from obtaining information necessary to run dictionary attack. This approach, too, is effective, but not complete. In workstation login systems, an adversary needs cryptographic hashed versions of passwords. He applies the same hash function to potential passwords in a dictionary, and sees if the result matches with the hashed version. The hashed passwords are stored on a workstation, for example, in /etc/passwd on UNIX and in \WINNT\SYSTEM32\CONFIG\SAM on Windows NT. These files can be made readable only to system administrators (e.g., through shadow password technique in UNIX). However, they still are available for an adversary in many ways: through a Trojan horse, through physical access, by exploiting bugs of daemons that run with administrative rights, by exploiting bugs of CGI programs, etc.

In a network authentication system, Kerberos, an adversary needs a plaintext-
ciphertext pair that is encrypted with a password. Such ciphertext is available as a message called authentication service reply, or AS.REP. This message is sent by Kerberos Key Distribution Center, or KDC, to a user in an authentication step, and is encrypted with a key derived from a user’s password. The plaintext of this message has some known text, such as, the user’s name and realm. Therefore, if the adversary can obtain an encrypted AS.REP, he can launch a dictionary attack. This message can be obtained through interception on a network between the KDC and the user’s workstation, or by asking the KDC to generate one. When requested for AS.REP, Kerberos IV does not check the identity of the client, so it gives this message up to anybody who requests it [95]. This has been addressed in Kerberos V, in which a user has to prove that he knows the password by sending an encrypted timestamp to the KDC (preauthentication step) [59]. However, the adversary can still obtain the message by sniffing the network.

Thus, neither approach is complete, and there is strong motivation to get rid of passwords. One possible way is to replace them with smartcard authentication.

## 2.3 Secure Hardware

To solve the problems I stated in the previous sections, researchers have suggested taking advantage of secure hardware.

The importance of physical security is discussed by Yee in his Ph.D. thesis: “Physical security is a central assumption ... without this foundation even the best crypto system or the most secure kernel will crumble.” He suggests using secure hardware as a foundation of secure operating system, and developed operating system primitives, such as a zero-knowledge authentication protocol [108]. His approach differs from mine in two ways: (1) Yee built a new system around secure hardware. In contrast, I
integrate secure hardware into existing infrastructures for minimum deployment cost.

(2) Yee used secure coprocessors exclusively. In contrast, I use smartcards as well as secure coprocessors.

Application of secure hardware for key storage is suggested by Checkley: “By its very nature, software is insecure and does not offer a secure hiding place. No matter how much it is disguised, a key held in software is always ultimately accessible to a computer professional. Tamper resistant hardware should be used to store keys securely” [18].

Necessity of secure hardware integration to Kerberos is argued by Bellovin and Merritt: “Support for special-purpose hardware should be added such as the keystore” [6].

Here I introduce two types of secure hardware: smartcards and secure coprocessors.

2.3.1 Smartcards

A smartcard is a plastic card with a microprocessor chip implanted in it. It is considered to be able to protect the data and the computation for the following reasons:

- Physical protection

Physical protection of smartcards makes it hard (although not impossible) to obtain data from the smartcard. Layers of material conceal the microchip on a smartcard: black epoxy resin, a passivation layer (usually silicon oxide or nitride), and a polyimide layer [61]. These layers make invasive attack (which destroys a smartcard) difficult, as they are hard to remove without destroying the data. They protect against non-invasive attack (which does not destroy a smartcard) as well, as they prevent information from leaking.
• Integrated circuit

Integrated circuit design of a smartcard makes it very difficult to probe the communication between the devices in it. A smartcard microchip usually has all of the devices (e.g., a processing unit, ROM, RAM, I/O, and cryptographic coprocessors) integrated into it, instead of having them separated and connected through buses [39].

• Physical possession and PIN protection

A smartcard is easily carried by a user, a significant security advantage over a larger computer. A smartcard is physically separated from the outside world most of the time. It is accessible by outsiders only when it is connected to a computer and is supplied with power. Therefore, it has a very short temporal window of vulnerability. Even if it is stolen by an adversary, if it is properly protected by a Personal Identification Number, or a PIN, the adversary cannot use it, nor access the information in it. In contrast, a desktop computer cannot be carried, and physical access by outsiders is much harder to control. Furthermore, it is usually connected to the Internet. Thus, its window of vulnerability is 24 hours a day. A laptop computer can be carried, too, but the user cannot carry it all the time as he might a smartcard.

• Restricted Interface

A smartcard usually exports a minimal set of interfaces to the host to avoid exporting flawed interfaces that lead to vulnerability. For example, it usually does not export interfaces to read keys out of it.

The history of smartcards is long. The idea goes back to late 1960s, when inventors filed patents on it in Europe and in Japan: Jurgen Dethloff and Helmut Grotrupp in
1968, Kunitaka Arimura in 1970, and Roland Moreno in 1974 [84]. Large-deployment started in 1980s. Telephone cards, health cards, and token cards for satellite TVs have been quite successful in Europe. In addition, smartcards are used as authentication modules in GSM phones over the world.

Many think that the next killer application of smartcards is the electronic commerce application, but I believe otherwise: smartcards have much more potential in the information technology business. E-commerce users seem quite happy with current infrastructures, such as credit cards, thus leaving little room for smartcards to thrive. In contrast, in the information technology field, there is a growing fear of hackers attacking personal information, and as this dissertation shows, smartcards can significantly improve this situation. As support for smartcard reader drivers is maturing [20], and operating system giants such as Microsoft and SUN Microsystems are starting to put smartcard support in their flagship operating systems, I believe the time is right to start wide deployment of smartcards in this area.

### 2.3.2 Secure Coprocessors

A secure coprocessor is a tamper-resistant computational and storage device, typically in a form of a PCI card or a PCMCIA card. Compared with smartcards, secure coprocessors typically are larger, faster, more expensive, and take more thorough approach for physical security, as follows [108]:

- **Protection against penetration**

  A secure coprocessor detects penetration attempts with thin nichrome wires wrapped around it. The wires, when broken by the penetrator, work as a sensor and cause emergency action by the coprocessor: it erases memory content in its non-volatile memory.
• Protection against chemical attack

A secure coprocessor also detects chemical attack, that is, an adversary tries to dissolve the epoxy that covers the wires. To counter this attack, the epoxy is made chemically “harder” than the sensor wires so that the solvents will destroy the wires before the epoxy is removed.

• Protection against low temperatures

Another attack is low temperature attack, that is, an adversary freezes it to disable the battery, and then extract the contents of the memory. The secure coprocessor detects this attack with a low temperature sensor.

• Protection against high-powered laser

Yet another attack is to use a high-powered laser to disable the sensing circuit. The secure coprocessor counters this attack with added alumina or silica in the epoxy. The metal generates heat when it absorbs the laser emission, and breaks the wires.

IBM Research has developed generations of secure coprocessors: Abyss [103, 105], Citadel [106, 80], and their first product, the IBM 4758 [94, 49]. Tygar and Yee used the Citadel prototype to develop operating system support for secure coprocessors [97, 108].

Similar to smartcards, secure coprocessors await killer applications. This dissertation shows that they can improve the security of modern computer systems. They are especially effective in the server side of cryptographic protocols, which demands great computational power.

Table 2.1 summarizes the difference between the smartcard and the secure coprocessor.
<table>
<thead>
<tr>
<th></th>
<th>Speed</th>
<th>Size</th>
<th>Cost</th>
<th>Physical Protection</th>
<th>Personalization</th>
<th>Used in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smartcard</td>
<td>Limited</td>
<td>Limited</td>
<td>Cheap</td>
<td>Strong</td>
<td>Yes</td>
<td>Client</td>
</tr>
<tr>
<td>SCC</td>
<td>Fast</td>
<td>Large</td>
<td>Expensive</td>
<td>Very Strong</td>
<td>No</td>
<td>Server</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison between Smartcard and Secure Coprocessor

2.4 Kerberos

Kerberos is one of the most widely used authentication systems of today, as it is secure, it scales, and it is standardized. It is a network authentication system based on the Needham-Schroeder protocol [77]. It is used in many universities to protect their computer network environments (Carnegie-Mellon, Cornell, MIT, Stanford, the University of Michigan, to name a few), and is part of products of many corporations (Sun, Microsoft, Cisco, IBM, Oracle, etc.) Microsoft Windows 2000 embraces it as a basic network authentication method. It is integrated into many network applications to provide authentication: UNIX login, ftp, telnet, PAM, SSH, AFS, DFS, and so on. Its security has been thoroughly analyzed [6, 59], and it scales quite well. For example, three replicated Kerberos servers at the University of Michigan serve 180,000 users. It is also quite portable; both the clients and the servers run on almost any UNIX or Windows systems.

That said, it has security limitations, too; it suffers from the two problems I introduced. To explain this, I briefly introduce the Kerberos protocol without details. Interested readers are advised to access articles [12, 95, 60, 58].

Kerberos provides two main services: Authentication Service (AS) and Ticket Granting Service (TGS). They are served by a trusted third party in Kerberos, i.e., Kerberos Key Distribution Center, or KDC. I introduce only the former service here, but the following discussion is applied to the latter as well. Figure 2.1 shows a
simplified representation of the AS protocol \(^1\), in which a Kerberos principal, Alice, is trying to prove her identity to the TGS.

1. Alice \(\rightarrow\) KDC  Alice, TGS, nonce, {time}\(K_A\)
2. KDC \(\rightarrow\) Alice  \{K_{A,TGS}, nonce\}\(K_A\), \{A, TGS, K_{A,TGS}\}\(K_{TGS}\)
3. Alice \(\rightarrow\) TGS  \{time\}K_{A,TGS}, \{A, TGS, K_{A,TGS}\}\(K_{TGS}\)

Alice (A)  Kerberos principal (user)
TGS  Ticket Granting Server
\(K_A, K_{TGS}\)  Keys of Alice and TGS
\(K_{A,TGS}\)  A session key shared between Alice and TGS
time  A timestamp that guarantees freshness of a message
nonce  A random nonce that proves the KDC knows Alice’s key

Figure 2.1: Kerberos Authentication Service Protocol

In step 1, Alice informs the KDC that she wants to be authenticated to prove her identity to the TGS. In addition to these basic messages, Kerberos V5 defines a preauthentication message, \{time\}\(K_A\). This proves that Alice knows her key. Preauthentication has been added to prevent an adversary from easily obtaining a plaintext - ciphertext pair for dictionary attack.

In step 2, the KDC verifies the preauthentication data, generates a TGS session key \(K_{A,KDC}\), and replies to Alice with an authentication service reply, or \(AS_{-REP}\), and a ticket granting ticket, or \(TGT\). The former provides the session key to Alice, and the latter provides the session key to the TGS.

In step 3, Alice sends the TGS an authenticator, \{time\}\(K_{A,TGS}\), to prove that she knows the session key, and the TGT.

The security limitations with this protocol are as follows:

- Vulnerability to untrusted code in the client

\(^1\)This is the AS protocol in Kerberos 5. AS in Kerberos 4 is slightly different, but the same discussion pertains.
cause major damage to Alice. (1) He can steal her password through a Trojan
horse, by sniffing a keyboard typing activity, or by finding it in memory. (2) He
can steal her key, $K_A$, which is derived from her password. (3) He can steal the
session key, $K_{A,TGS}$, and the TGT, to impersonate her.

This obviously is a serious threat to Alice because with her password or key,
the adversary can access any resource on her behalf.

- Vulnerability to untrusted code in the server

If the adversary compromises the server host (i.e., the host the KDC and the
TGS run on), the situation is even worse. To serve as a trusted third party, the
KDC has the keys of all the principals, including Alice and the TGS. Therefore,
once the adversary compromises the KDC, it can steal all the keys, and can
access all the resources on behalf of anybody.

- Vulnerability to dictionary attack on Alice’s password

The adversary can run dictionary attach in search of Alice’s password. Alice’s
key, which is derived from her password, is used to encrypt $AS\_REP$,
$\{K_{A,TGS}, n\}K_A$. As the message contains known texts, such as her name and
her realm (although these are not shown in this simplified representation), the
adversary can run dictionary attack to this message. He can obtain this message
by sniffing the network between Alice and the KDC.

I have developed solutions to these problems utilizing secure hardware. That is,
$K_A$ and $K_{A,TGS}$ are stored in the smartcard in the client side, and they are stored in
a secure coprocessor in the server side. They are reported in Chapter 3 in details.

In this dissertation, a word “Kerberos” refers to Kerberos V5 [60, 58], unless
specified otherwise.
2.5 Cryptographic File System

Cryptographic File System, or CFS, is a file system that encrypts files stored in it. It was developed by Matt Blaze [9]. It protects files from storage theft; even if an adversary steals a hard disk that contains secret files, he cannot read them, as the files are encrypted under a key that only the user knows. It is transparent to the user; because it carries out cryptographic operations in the file system level, i.e., below the application level; the user sees the files in the clear, as though they are in a normal file system. Its performance overhead is small, as CPUs are very fast nowadays. It is also quite portable, as it is written as a NFS daemon; it runs on almost all UNIX systems.

However, it has security limitations derived from the two problems I repeatedly mention: trusted computing and passwords. Before discussing the problems, I introduce the key management system of CFS. When a user creates a directory with a command `mkdir`, she types in a 16 byte (or longer) password. The password is converted into a key \(^2\), which I call a directory key. CFS stores this key in memory, and uses it to encrypt a file on write, and to decrypt a file on read.

Now I discuss the security limitations of CFS:

- Vulnerability to untrusted code in the host
  
  If an adversary compromises the host computer while the user is using CFS, he can find the directory key in memory, and probably the password, too. Then the adversary can decrypt all the files in the directory.

- Vulnerability to dictionary attack on the user’s password
  
  CFS relies on a user chosen password for key generation, and therefore is vul-

\(^2\)Into three keys, in fact, but I omit the detailed description here.
nerable to dictionary attack. CFS is at most as secure as the passwords. As described in Section 2.2, the adversary must obtain some information to run dictionary attack on. In CFS, the adversary can use a plaintext-ciphertext pair. This is available through host compromise or physical access. Encrypted files stored in a hard disk are the ciphertext, and the adversary usually can distinguish the right plaintext because it usually includes human readable text, e.g., ASCII text.

I have developed a countermeasure that utilizes secure hardware. That is, to store a user’s master key in a smartcard, use it to generate a file key, which is used to encrypt a single file, instead of a directory. This is reported in Section 4.2 in details.

2.6 AEGIS: Secure Bootstrap Process

AEGIS is a secure bootstrap process developed by Arbaugh et al. Its goal is to provide a trusted foundation on a computer system. As described in Section 2.1, a modern computer system cannot be trusted because of the lack of physical security and software bugs. One way of addressing this problem is to enforce integrity of a computer system. A system is said to possess integrity if no unauthorized modification has been made to it (Denning defines integrity similarly for communication in [24]). AEGIS verifies integrity of a personal computer at boot time, through a technique called code signing.

Code signing works as follows:

1. A user generates a hash, $H$, of a program.

2. The user signs $H$ with her private key. The signed hash is called a Message Authentication Code, or a MAC.
3. She stores the MAC in a write-protected device.

4. Before the program runs, she calculates its hash and checks whether it matches the stored MAC decrypted with the public key.

When two hashes match, it is almost certain that the program that runs at step 4 is the same as the one in step 1, therefore, it has not been modified.

Code signing for application software is widely used, for example, Tripwire [54] and Authenticode [99], but they do not provide OS integrity. Therefore, they are still vulnerable to OS compromise, which is possible through physical attack and software bug exploitation. Phrack magazine reports a method to fool Tripwire with a loadable kernel module that redirects system calls [41].

In contrast, AEGIS is an OS integrity checker, whose basic idea is as follows. A modern computer system is often organized with layers to limit complexity. Each layer verifies the layer above it before booting (i.e., passing control to) the layer above with code signing. If the root of this chain of checks is trustworthy (AEGIS ensures this by storing the root of the chain, i.e., BIOS, in read-only memory), and the checks are successful, the integrity of a computer system is assured at the boot time. This leaves the possibility of attacks after a system is booted, but it is out of AEGIS’s scope.

Unlike Kerberos and CFS described above, AEGIS tolerates physical attacks, as it has secure hardware: the BIOS in read-only memory. Therefore, the purpose of my extension to AEGIS is different from on Kerberos and CFS. It is to achieve personalization, authentication, and flexibility. In AEGIS, system administrators have total control over what code to be run. My smartcard integrated system, sAEGIS, hands this control to a user.

This work is reported in Chapter 5.
2.7 Research Aiming at Similar Goals

2.7.1 Type-Safe Languages

The majority of software bug exploits introduced in Section 2.1 take advantage of memory pointer related bugs, e.g., buffer overflow. Some languages, such as Modula-3, Standard ML, and Java, are type-safe, i.e., they check types strictly, prohibit casting, and prohibit pointer arithmetic operations, thus preventing pointer related bugs. Operating systems have been built over such languages (e.g., SPIN is written with Modula-3 [8]). If all software is written in type-safe languages, the computer is more secure than most current ones because it does not have the pointer related problems. Yet, even though it has been proven possible, it is not practical to rebuild an entire computer environment from scratch with type-safe languages because the cost is too great.

2.7.2 Secret Sharing

Another approach to the problems stated in Section 2.1 is the secret sharing mechanisms. Researchers who have found that computer systems can be compromised or intruded [45, 107] have developed the mechanisms that split a secret (e.g., key) into $n$ pieces, and distribute them to $n$ computers. To recover the secret, at least $k$ ($k < n$) computers must reveal their pieces. Therefore, an adversary compromising fewer than $k$ computers cannot obtain the secret.

Using a secret sharing approach raises the bar of the system’s security by forcing intruders to have to break into $k$ computers to compromise the entire system. In contrast, the secure hardware approach makes the compromise of only a single computer much more difficult.
2.7.3 Sandboxing

The sandbox approach limits the power of untrusted software. For example, the Java sandbox system prohibits Java applets from accessing filesystems, and from connecting to arbitrary computers [36]. Goldberg et al. developed a system to restrict the use of system calls by helper applications in a browser [35]. Lepreau et al. developed a recursive virtual machine (RVM) model, which consists of nested virtual machines. Operating system resources are “donated” by the parent VM to child VMs; the children cannot access the parent’s resources unless they are donated [65]. Therefore, a compromised child VM can do no harm to its parent VM.

The sandbox approach is similar to my approach in that both limit the power of untrusted software in terms of the operations it can execute and the resources it can access, and minimize the damage it can cause when it is compromised. For example, Goldberg’s approach prevents an application from accessing certain files; My approach prevents host-side software from accessing certain keys in secure hardware. However, my approach is more thorough because I do not trust even the operating system kernels. Furthermore, my approach counters physical attack. The main drawback of my system is that it requires specialized hardware.

2.7.4 Top-Down Secure Operating System

Some researchers have taken top-down approaches to build secure distributed computer systems.

In the Dyad system [97], Tygar and Yee take a top-down approach in building a secure computer system. Unlike most security protocols, they do not assume the integrity of the components of computers such as operating systems. To address this problem, they embrace secure hardware (secure coprocessor) and build an operating system around it.
Another related, top-down approach is described by Lampson et. al. [63], who develop a theory of authentication for distributed systems based on an access control model. They build tools necessary for secure systems, such as encrypted channels, bootstrapping, naming, and program loading. Accompanying the design of these tools are formal proofs of their security. Finally, they build an operating system called TAOS to take advantage of the tools.

Both Dyad and TAOS take top-down approaches: they start with a well-developed theoretical framework, embrace secure hardware to support the theory, and then build operating systems based on them.

Although these approaches are substantive and technically sound, they are not practical for most existing computer environments because they build new operating systems from scratch. I take a more pragmatic and experimental approach and build from the bottom-up for rapid prototyping, implementation, and deployment. I employ currently available, secure, inexpensive hardware in the form of commercial smartcards and secure coprocessors, integrate them with prevalent standards, and fit them with minimal effort into existing distributed computer systems.
Chapter 3

Secure Hardware Integration with Kerberos

As discussed in Section 2.4, Kerberos is a well accepted and important network authentication system, but suffers from the two problems: untrusted computing and passwords. I argue that these problems can be solved by integrating secure hardware into Kerberos, without modifying the basic part of Kerberos. This chapter reports the detailed design, the security consideration, the implementation, and the performance of such a solution. Section 3.1 reports smartcard integration with Kerberos V5 client. Section 3.2 reports secure coprocessor integration with Kerberos V5 server. Finally, the smartcard - Kerberos integration is extended to a remote smartcard in Section 3.3. To keep the developed systems easy to deploy, the client side modifications (Section 3.1 and 3.3) do not affect the server, and the server side modification (Section 3.2) does not affect the client.

3.1 Smartcard Integration with Kerberos V5 Client

3.1.1 Introduction

As argued in Section 2.4, Kerberos is vulnerable to client host compromise and dictionary attack. To solve this problem, I have integrated a smartcard into the Kerberos
client. In this section, I report the design, security consideration, implementation, and the performance of the project.

3.1.2 Design

Design Goals

I set the design goals as follows, from the discussion in Section 2.4.

1. Security critical keys in a smartcard

Security critical keys, namely, keys that should be protected especially carefully, are stored in a smartcard, and never leave. The keys are therefore protected from an adversary who can compromise the user’s workstation. In the Kerberos Authentication Service (AS) and Ticket Granting Service (TGS) protocols, shown in Figure 3.1, two keys are the most critical: a user’s master key ($K_A$) and a session key between the user and TGS ($TGS$ session key, or $K_{A,TGS}$). The former is the most important, as this proves the user’s identity; if an adversary obtains this key, he can impersonate the user until the key is changed. The latter is also important, as this proves that the user has successfully gone through Authentication Service; if an adversary obtains this key, he can impersonate the user for the key’s lifetime (which is 8 hours by default). Therefore, these two keys should be stored in a smartcard.

2. Decryption of $\text{AS.REP}$, encryption of $\text{TGS.REQ}$, and decryption of $\text{TGS.REP}$ in a smartcard

As shown in Figure 3.1, these cryptographic operations involve either the master key or the TGS session key. Because these keys are stored in a smartcard by Goal 1, the smartcard must carry out these operations. These operations are
DES, and many smartcards are capable of this.\footnote{Or claim to. Many say they offer DES but they in fact do not. We discuss this further in Section 3.1.6}

3. Random bit string for master key

A user’s master key, which is derived from a password in the original Kerberos, is replaced with a randomly generated bit string. This prevents dictionary attack.

4. No modification to KDC or TGS

KDC and TGS, the servers in Kerberos AS and TGS, are not modified. This allows seamless deployment of the developed system.

Protocol

\begin{itemize}
\item AS
\item Workstation $\rightarrow$ KDC: Alice, TGS, nonce
\item KDC $\rightarrow$ Workstation: $\{K_{A,TGS}, \text{nonce}\}K_A, \{A, \text{TGS}, K_{A,TGS}\}K_{TGS}$
\item TGS
\item Workstation $\rightarrow$ TGS: Bob, \{time\}$K_{A,TGS}, \{A, \text{TGS}, K_{A,TGS}\}K_{TGS}, \text{nonce}'$
\item TGS $\rightarrow$ Workstation: $\{K_{A,B}, \text{nonce}'\}K_A, \{A, B, K_{A,B}\}K_B$
\end{itemize}

Alice (A), Bob (B) Kerberos principals (users)
KDC Key Distribution Center
TGS Ticket Granting Service
Workstation Alice’s workstation
$K_A, K_B, K_{TGS}$ Keys of Alice, Bob, and TGS
$K_{x,y}$ A session key shared between x and y
time A timestamp that guarantees freshness of a message
nonce A random nonce that proves KDC knows Alice’s key
nonce’ A random nonce that proves TGS knows $K_{A,TGS}$
Smartcard Alice’s Smartcard

Figure 3.1: Kerberos Authentication Service and Ticket Granting Service without Smartcard

Figure 3.2 presents the Kerberos AS and TGS with smartcard modifications.\footnote{I omit preauthentication in this figure. This is to save one smartcard call. This should not affect the security of the protocol. Preauthentication was introduced to make off-line dictionary attack difficult. My scheme invalidates dictionary attack by replacing a password with a random key.}
AS
1. Workstation → KDC Alice, TGS, nonce, \{time\}K_A
2. KDC → Workstation \{K_{A,TGS}, nonce\}K_A, \{A, TGS, K_{A,TGS}\}K_{TGS}
   2.1. Workstation → Smartcard \{K_{A,TGS}, nonce\}K_A
   2.2. Smartcard → Workstation nonce
TGS
3.1 Workstation → Smartcard time
3.2 Smartcard → Workstation \{time\}K_{A,TGS}
3.3. Workstation → TGS Bob, \{time\}K_{A,TGS}, \{A, TGS, K_{A,TGS}\}K_{TGS}, nonce’
4. TGS → Workstation \{K_{A,B}, nonce’\}K_A, \{A, B, K_{A,B}\}K_B
   4.1. Workstation → Smartcard \{K_{A,B}, nonce’\}K_{A,TGS}
   4.2. Smartcard → Workstation K_{A,B}, nonce’

Figure 3.2: Kerberos Authentication Service and Ticket Granting Service with Smartcard

Interaction between the workstation and KDC and TGS, i.e., steps 1, 2, 3, and 4, are identical to the original protocol (Figure 3.1). Six additional steps, 2.1, 2.2, 3.1, 3.2, 4.1, and 4.2 are added to access the smartcard: (2.1) When the workstation receives AS.REP, it does not decrypt it. Instead, it sends it to the smartcard. (2.2) The smartcard then decrypts it, returns it in the clear to the workstation, except the TGS session key. The key must stay in the smartcard, so it is masked from AS.REP. (3.1, 3.2) The workstation sends the timestamp to the smartcard to get it encrypted with the TGS session key. (4.1, 4.2) The workstation sends TGS.REP to the smartcard to get it decrypted with the TGS session key.

This protocol satisfies the goals stated above: (G1) the master key and the TGT session key never leave the smartcard, (G2) all the cryptographic operations that involve these two keys are carried out in the smartcard, (G3) the master key can be random bit string, and (G4) KDC and TGS are not modified.
3.1.3 Security Discussion

Here I discuss the security of the smartcard integrated Kerberos (sc-Kerberos) and the original Kerberos (Kerberos).

Model

I start with constructing a model of a Kerberos realm. The model consists of the following participants:

Alice  A Kerberos principal in this realm. She wants to establish a session key with Bob.

Bob  Another Kerberos principal.

Host  Alice’s workstation.

Smartcard  Alice’s smartcard.

KDC  Key Distribution Center of this realm.

TGS  Ticket Granting Server of this realm.

Mallory  An adversary.

Network  A network that connects Alice, Bob, KDC, and TGS.

Threats

I make the following assumptions in my model.

1. Mallory can compromise a host.

From discussion in Section 2.1, Mallory can read and modify any information on the host.

33
2. Mallory can change behavior of the host, but it is difficult.

By Assumption 1, Mallory is able to install a Trojan horse in the host, which, for example, steals the password or the session key and the ticket. However, I assume this attack is difficult 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 [54].
- It becomes even harder if Alice uses hardware based integrity checker, such as AEGIS [3] and sAEGIS (Chapter 5).

3. Mallory can compromise the network.

Mallory can obtain and alter any data on the network.

4. Mallory cannot compromise the smartcard.

From discussion in Section 2.3.1, Mallory can neither read nor modify any information in the smartcard. She cannot influence the behavior of the smartcard, either.

5. Mallory cannot compromise Bob, KDC, and TGS.

The other principals in this realm are assumed secure. Section 3.2 discusses Kerberos server security in details.

6. Alice’s password is weak.

From discussion in Section 2.2, I assume that Alice’s password is vulnerable to dictionary attack.
7. Key in smartcard is random.

I assume that Alice’s key stored in the smartcard is random and therefore cannot be guessed by dictionary attack.

8. Cryptographic operations are strong.

My 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 [30]. DES should be replaced with DESX or triple-DES [89] in the future.

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

**Attack**

**Master Key Theft**

In Kerberos, Mallory can steal the master key by compromising the host (which is possible by Assumption 1).

In sc-Kerberos, Mallory cannot. The key is in the smartcard, which cannot be compromised (Assumption 4). The key cannot be stolen from KDC, either (Assumption 5).

**TGT Theft**

In Kerberos, Mallory can steal the TGT and the TGS session key by compromising the host. Mallory can then use them to receive obtain service tickets as Alice. Because this lets Mallory to impersonate Alice temporarily (for TGT lifetime, which is 8 hours by default), this is a severe security threat.

In sc-Kerberos, Mallory cannot. The TGS session key is in the smartcard, which cannot be compromised (Assumption 4). The key cannot be stolen from KDC, either (Assumption 5).
Dictionary Attack

In Kerberos, Mallory can obtain Alice’s password by obtaining an encrypted AS.REP by sniffing the network (possible by Assumption 3), and running dictionary attack on it. As Alice’s password is weak (Assumption 6), dictionary attack succeeds and Mallory obtains the password.

In sc-Kerberos, Mallory cannot run dictionary attack, as the key stored in the smartcard is random (Assumption 7), and brute force attack is impossible (Assumption 8).

On-Line Attack

In Kerberos, a Trojan horse installed on the host (possible by Assumption 2) can steal Alice’s password. For example, a wrapper around the kinit program can send the password to Mallory.

In sc-Kerberos, a Trojan horse cannot steal the master key, as it is in the smartcard. However, it still can steal the PIN of Alice. With the PIN in hand, Mallory can make the smartcard do whatever she wants it to do. This means that Mallory can use the master key while the smartcard is attached to the host. However, the window of vulnerability is much smaller with sc-Kerberos (while the smartcard is attached), than with Kerberos (all the time after the attack). In addition, sc-Kerberos can be made more secure with additional hardware. If the smartcard reader or the smartcard itself has a PIN pad, Alice can enter a PIN without the risk of being snooped by Mallory. If the host, the reader, or the smartcard has an output device that indicates transaction between the host and the smartcard (e.g., LED blinks when data is transmitted) Alice can tell when the host is using the smartcard even though she did not instruct it to.
3.1.4 Implementation

Implementation of the smartcard enhanced Kerberos protocol described in Section 3.1.2 is reported in this section.

The smartcard used is Cyberflex Access from Schlumberger, a Java programmable card. It provides native *cipher block chaining* (CBC) mode DES encryption. CBC is helpful, as the messages processed are up to ∼200 bytes long.

Host side code base is Heimdal version 0.3b. The development platforms are Linux 2.2, OpenBSD 2.7, and Solaris 2.7. The sc7816 library developed at CITI [85] is used to communicate with smartcards.

Four portions of the code base were modified: the Kerberos library (libkrb5.a), the DES library (libdes.a), the AS client (kinit), and the TGS client (tcp_client). Modification to each portion is detailed below.

**Addition of an encryption system to the Kerberos library**

Kerberos V5 eases replacement of encryption systems through a look-up table [58]. The look-up table associates an encryption type to cryptographic functions, such as encryption, decryption, checksum functions, and data structures (e.g., a key structure). Adding a new encryption system is done by adding an entry to the look-up table.

There are several encryption system types defined in the RFC-1510 [58] and implemented in Heimdal including:

- No encryption

---

3 In the initial implementation of this project, STARCOS version 2.1 from Giesecke & Devrient was used. It performed very well. However, as the project grew, I started to need programmability of Java Card, which STARCOS did not possess.

4 My initial implementation, which carries out AS but not TGS on smartcard, was based on MIT Kerberos V5 version 1.0.5 and 1.2.1.
• DES in CBC mode with a CRC-32 checksum (des-cbc-crc)

• DES in CBC with MD5 (des-cbc-md5)

A new encryption system, DES in CBC with MD5 with a smartcard, is created. Its entry (des-cbc-md5-sc) added in the look-up table. The entry is defined in crypto.c (Figure 3.3).

```c
krb5_cryptosystem_entry
mit_des_md5_sc_cryptosystem_entry {  
  /* encryption type */ ENCTYPE_DES_CBC_MD5_SC;  
  /* encryption function */ DES_CBC_SC_encrypt_null_ivec();  
  /* keyed-hash function */ NULL;  
  // Other members are identical to des-cbc-md5
};
```

Figure 3.3: DES-CBC-MD5-Smartcard Cryptosystem Entry

DES_CBC_SC_encrypt_null_ivec() is a new function that uses a smartcard for encryption. The pointer for a keyed-hash function is set to NULL, as I decided to use the md5 hash function instead of the md5des hash function (to save one smartcard call). The other members of the entry are identical to the ones of the entry for DES in CBC with MD5 (des-cbc-md5).

MD5 is selected over CRC, the default hash function in Kerberos, because the implementation of des-cbc-crc in Kerberos V5 has a bug. The Kerberos V5 specification defines the *initialization vector* (IV) to be 0 [58], but the implementation uses the key as the IV. This error can not be fixed easily because Kerberos V5 is already deployed widely and several commercial offerings use the key as the IV as well. Cyberflex is flexible, and it can implement this mode by reading the key file. However, I decided not to because reading a key from a key file is not a natural operation; keys in key files are meant to be used for cryptography, but not to be read. Even though
it is possible with Cyberflex, it may not be the case with other smartcards (e.g.,
STARCS does not allow this). It also adds cost for reading a key from EEPROM.

To my relief, des-cbc-md5 uses 0 as the IV, complying with the RFC. Both
Heimdall and MIT Kerberos now support des-cbc-md5.

Modification of DES library

DES_CBC_SC_encrypt_null_ivec() calls the DES_CBC encryption function in cbc_enc_sc.c.
I created a new DES_CBC function sc_encrypt(), which calls a DES function in a
smartcard through the sc7816 library. Cyberflex can handle up to 248 bytes in a
DES CBC call. If a message encrypted or decrypted by a smartcard is more than
248 bytes, the message is divided into two pieces, decrypted in a smartcard piece by
piece, and combined into one message in the workstation.

Modification of kinit

An authentication service client, kinit, is modified so that it carries out AS with a
smartcard when invoked with -C or -D option. -D stores a TGT session key in a smart-
card. An encryption system is chosen in an argument for the authentication function
krb5_get_init_creds_password(). kinit.c is modified so that it does not request
a password from the user, and it specifies the encryption type des-cbc-md5-sc.

Modification of tcp_client

A ticket granting service client, tgs_client, is modified so that it carries out TGS
with a smartcard when invoked with -C.
Smartcard Applet

The smartcard part of this protocol is written in Java. Its interface includes the following. *Application Protocol Data Units (APDUs).*

- decrypt with master key
  
  03 10 00 00 len data

- decrypt with master key, and store TGT session key in smartcard
  
  03 14 00 00 len data

- decrypt with TGT session key
  
  03 18 00 00 len data

- encrypt with TGT session key
  
  03 18 00 01 len data

- encrypt with key derived from TGT session key
  
  This is for md5des checksum calculation.
  
  03 18 00 02 len data

- read principal - key table
  
  A smartcard can hold more than one key (two in the current implementation). It stores a table that maps a principal name to a key number. This APDU sends the table to the host.
  
  03 20 00 00 00

  An example key table is as follows.

---

5Readers not familiar with smartcard APDUs are advised to consult the ISO 7816-4 specification [50] or Guthery and Jurgensen’s book [39].
• set key number

This command selects the key used for AS.

03 30 00 00 01 key_num

3.1.5 Performance Evaluation

Here I evaluate the performance of sc-Kerberos.

Overview

I measured the performance of AS and TGS in sc-Kerberos. AS takes 3.49 seconds on average and TGS 3.14 seconds with small fluctuation (about 0.1 sec). kinit and tcp_client were executed five times and the average round-trip time was taken. Cyberflex communicates with the host at 38.4 Kbps. I analyze performance in detail in the following sections.

AS Time Line

Table 3.1 shows the time line of kinit. In the total time 3.49 sec, smartcard related tasks clearly dominate, taking 96% (3.36 sec) of the round-trip time. With an 8-bit data path and a 3.5 MHz clock, a smartcard is much slower than a workstation.

AS Breakdown

Table 3.2 shows how much time is spent in each part of AS. Time is in seconds. These numbers are obtained by timing individual operations.

The result shows that searching and masking the session key, and DES decryption are the bottlenecks. The former operation scan the decrypted ticket, finds the session
<table>
<thead>
<tr>
<th>event</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>0.00</td>
</tr>
<tr>
<td>open card start</td>
<td>0.06</td>
</tr>
<tr>
<td>reset card start</td>
<td>0.31</td>
</tr>
<tr>
<td>table read start</td>
<td>0.57</td>
</tr>
<tr>
<td>set key number start</td>
<td>0.78</td>
</tr>
<tr>
<td>ASREP decryption start</td>
<td>0.89</td>
</tr>
<tr>
<td>ASREP decryption end</td>
<td>3.42</td>
</tr>
<tr>
<td>kinit end</td>
<td>3.49</td>
</tr>
</tbody>
</table>

Table 3.1: Authentication Service with Smartcard Time Line

<table>
<thead>
<tr>
<th>part</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>searching and masking the session key</td>
<td>1.22</td>
</tr>
<tr>
<td>DES decryption</td>
<td>0.95</td>
</tr>
<tr>
<td>APDU handling overhead (3 APDUs, 0, 1, 200)</td>
<td>0.48</td>
</tr>
<tr>
<td>card reset</td>
<td>0.26</td>
</tr>
<tr>
<td>scopen()</td>
<td>0.25</td>
</tr>
<tr>
<td>get response (3 calls)</td>
<td>0.10</td>
</tr>
<tr>
<td>file access</td>
<td>0.07</td>
</tr>
<tr>
<td>total</td>
<td>3.33</td>
</tr>
</tbody>
</table>

Table 3.2: Authentication Service with Smartcard Time Breakdown
key, stores it on card, and masks it. This is necessary when a user does not want the session key to be stored on the workstation. These two consume about 2/3 of the total time. The former is slow, as (1) it is a loop which goes through every byte of the message, (2) it invokes Util.arrayCompare() method for every step, (3) at the end of the loop, it executes two expensive operations DES_KEY.setKey() and Util.arrayCopy(), each of which takes about 0.1 second. DES is time consuming because it is a complicated algorithm.

**TGS Time Line**

Table 3.3 shows the time line of tcp_client. As in AS, in the total time 3.14 sec, smartcard related tasks clearly dominate, taking 96 % (3.01 sec) of the round-trip time.

**TGS Breakdown**

Table 3.4 shows how much time is spent in each part of TGS. Time is in seconds.

The result shows that DES decryption is the bottleneck.

Unfortunately, the bottleneck operations cannot be sped up without improving the smartcard itself. As a software developer, I can only hope there will be faster cards in the future.
<table>
<thead>
<tr>
<th>part</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES decryption of TGS_REP (240 byte)</td>
<td>1.02</td>
</tr>
<tr>
<td>DES encryption of authenticator (160 byte)</td>
<td>0.69</td>
</tr>
<tr>
<td>APDU handling overhead (2 APDUs)</td>
<td>0.63</td>
</tr>
<tr>
<td>card reset</td>
<td>0.26</td>
</tr>
<tr>
<td>scopen()</td>
<td>0.25</td>
</tr>
<tr>
<td>get response (3 calls)</td>
<td>0.12</td>
</tr>
<tr>
<td>total</td>
<td>2.97</td>
</tr>
</tbody>
</table>

Table 3.4: Ticket Granting Service with Smartcard Time Breakdown

3.1.6 Discussion

Related Work

Authentication with Smartcards

Several authentication protocols that use smartcards have been developed. For example, Rubin proposes one-time passwords [87], Shoup and Rubin propose session key distribution in the third-party setting [91], Leach proposes use of zero knowledge authentication [64], Wang and Chang propose use of public key authentication in smartcards [102], Bakker proposes an authentication protocol that can be used with smartcards with limited cryptographic functionality [4]. All of them concentrate on one-to-one authentication. This differs from my approach in that Kerberos is a network authentication protocol. In addition, I integrate a smartcard into a standard authentication protocol already in heavy use. Their protocols are all new, and among them, only Shoup and Rubin’s protocol has actually been implemented with a smartcard [52].

Smartcard Integration with Kerberos

Several researchers have suggested integrating smartcards into Kerberos, e.g., projects by Krawjeski et al. [53] and Looi et al. [68]. Their work differs from mine in that they did not implement authentication service with smartcards. Work by Gaskell et
al. implements this [33], but my work goes further, namely, ticket granting service with a smartcard.

**DES in Smartcards**

Many vendors claim that their smartcards support DES, but I had a very hard time getting a smartcard that meets my requirements: pure, unaltered DES. Here I list some of the DES-capable smartcards that let me down when examined closely.

To the credit of the vendors listed here, this list was created in 1999, when the U.S. export restriction law for strong cryptographic algorithm was still effective. It was logical for them to produce smartcards with crippled DES. Therefore, this list is not meant to criticize their decisions. It is meant to point out how smartcard software developers have suffered from the lack of clear statement about their cards’ functionalities. Because their advertisements claimed that their cards had DES, I purchased most of these smartcards only to discover that they in fact did not.

- **Schlumberger CryptoFlex**
  
  Only custom-made cards have DES.

- **Schlumberger MultiFlex**
  
  DES is available in the form of an internal authentication command, which returns only six of the eight bytes of output data.

- **IBM MFC**
  
  The smartcard encrypts a random number challenge presented by `SCT_CMD_AUTHENTICATE` command, but does not document a general-purpose DES interface.

- **MAOSCO MULTOS**
The card supplied with the developer’s kit encrypts with a fixed key, 0x41ad8223a90be2a1. According to the manual, “for security reasons,” DES uses a “known cryptographic key.” (!)

- General Information Systems OSCAR

The DES key is XOR’ed with a random number before it is used. According to their e-mail: “The keys are XOR’ed with a random number for security reasons.” While this may help secure the serial link between the terminal and the reader, it makes the card useless for enterprise security deployment.

- Gemplus GPK

The key size is limited to 40 bit, a flaw not shared by Kerberos.

Eventually I found smartcards that satisfy my needs in STARCOS and Cyberflex Access.

3.1.7 Section Summary

In this section, I pointed out two security problems of modern computer systems that are serious threats to Kerberos. I then developed countermeasures. I suggested a protocol that takes advantage of the secure features of a smartcard to enhance the security of Kerberos clients. The protocol is implemented with a Schlumberger Cyberflex Access smartcard, and with Heimdal. Performance evaluation shows the protocol runs reasonably fast.

The following two aspects highlight the value of this work.

Improvement to important software

As discussed in Section 2.4, Kerberos is a very important authentication system. The problems identified, client host compromise and reliance on passwords, are serious
security threats to Kerberos. The solution suggested in this section counters these problems.

The developed system relieves a user from inconvenience of repeatedly entering his password. As one smartcard holds multiple keys, once a user types in the PIN, the smartcard can carry out authentication for multiple realms.

First practical network authentication system for smartcards

Using a smartcard for computer authentication is not a new idea. As discussed above, there have been many authentication systems for smartcards proposed. However, this is the first network authentication protocol that can be integrated seamlessly into the current infrastructure, as it does not require modifications in the server side of Kerberos.
3.2 Secure Coprocessor Integration with Kerberos V5 Server

3.2.1 Introduction

The problem of trusted computing is a serious security threat for Kerberos servers, as well as for Kerberos clients. In fact, it is more serious for servers. A Kerberos KDC stores all the keys of the principals in the realm, and these keys will be revealed on KDC compromise. As a result, the adversary will be able to impersonate any user and access any protected resource. This attack is possible if the adversary has physical access to the host which runs the KDC, or obtains administrative access right on the host. The KDC stores its master key in a local file (/usr/local/var/krb5kdc/.k5.DOMAINNAME in MIT Kerberos V5) in the clear. All other keys are encrypted with the master key, and are stored in the disk, too. If the adversary can read the disk, he can obtain all the keys.

Therefore, it is essential to protect KDC as securely as possible, but as discussed in Chapter 1 and Section 2.1, it is quite difficult to achieve high security with current commodity computers. To solve this problem, I integrate a secure coprocessor, the IBM 4758, into a Kerberos KDC. The resulting KDC survives host compromise.

Discussion in this section can be applied for other Trusted Third Party (TTP) based protocols as well. For example, a certificate authority (CA) based authentication system has the same problem and similar countermeasures. Compared with a KDC, a CA has better characteristics when compromised because a CA stores public keys, but not private keys; the adversary cannot impersonate a user without changing the user’s key. However, he can still do so by crafting bogus certificates, so CA compromise is potentially quite damaging, too. CAs can be protected in the same manner as KDCs can.
The design, security consideration, implementation, and performance evaluation of the 4758 / Kerberos integration are presented in this section.

### 3.2.2 Design

#### Design Goals

To achieve host compromise survivability, there are two approaches. One is to implement an entire KDC in the 4758, and the other is to move keys and cryptographic operations to the 4758. I did not take the first approach, as it would limit the performance and scalability of the KDC. In terms of performance, the 4758 has a 66 MHz 80486 processor, significantly slower than modern personal computer processors. In terms of scalability, it has 8.5 KB of battery-backed RAM (BBRAM) and 1 MB of flash memory, allowing it to store many DES keys. However, storage in the host is always more abundant than storage in the 4758, and a Kerberos realm may require a huge number of keys, for example, at a university. Therefore, I decided to take the second approach. The following are the design goals.

1. Master key in the 4758

   The KDC’s master key is stored in BBRAM in the 4758 and never leaves.

2. Other keys stored in the host, encrypted with the master key. Used in the 4758.

   The other keys, TGS’s master key and principals’ master keys, are secure if they are stored in the 4758. However, storage limitation of the 4758 does not allow this, so these keys are stored in the host, encrypted with the master key. When the keys are needed, they are brought into the 4758, decrypted, used, and then removed from it.\(^7\)

---

\(^6\)90 MHz in Model 2

\(^7\)I can improve performance of the system by caching the keys. I did not try this optimization for the prototype; performance is not yet my primary goal.
3. Cryptographic operations in the 4758

To achieve Goals 1 and 2, all cryptographic operations that are associated with
the protected keys, i.e., decryption of keys, decryption of authentication data,
encryption of tickets, must be carried out in the 4758.

In addition to encryption and decryption, session key generation is carried out
in the 4758. This protects the session keys from an adversary. In addition, the
4758 can generate a better random number than a personal computer, as it has
a hardware seeded random number generator.

Protocols

Authentication Service

Figure 3.4 shows the Kerberos V5 authentication service (AS). Table 3.5 presents the
notation. The keys (the keys of Alice, TGT, and the KDC master key) are used in
the host. The master key is not shown in the figure, but is used to decrypt the other
keys. If the host is compromised, all the keys are revealed.

To solve this problem, I designed the protocol with the 4758 in Figure 3.5. All the
encryption, decryption, and random number generation are carried out in the 4758,
and no key is in the host in the clear.

AS Consistency Check

The 4758 generates the ticket and AS_REP only if the request is consistent, namely,
the following conditions are met:

- Key of Alice is used in preauthentication
- Alice is the client name in the ticket
- TGS is the server name in the ticket
Alice (A), Bob (B)  Kerberos principals (users)
KDC          Key Distribution Center
TGS          Ticket Granting Service
Workstation  Alice’s workstation
K_A, K_B, K_TGS Keys of Alice, Bob, and TGS
K_{x,y}      A session key shared between x and y
time         A timestamp that guarantees freshness of a message
nonce        A random nonce that proves KDC knows Alice’s key
nonce’       A random nonce that proves TGS knows K_{A,TGS}
Smartcard    Alice’s Smartcard

Table 3.5: Kerberos / 4758 Notation

Figure 3.4: Authentication Service without the 4758

The parties send additional information, such as message types, protocol version number, option flags, and start/expiration/renew-until time. I omit them in this figure because they are treated the same with or without the 4758.

Figure 3.5: Authentication Service with the 4758

Security critical tasks, e.g., en(de)cryption and random key generation, are moved from the host to the 4758. The host sends the 4758 the information needed for such tasks, e.g., the nonce sent by Alice, and the encrypted keys of Alice and TGS. The 4758 generates the TGS session key, puts it into AS_REP, encrypts it, and sends it back.
- Key of TGS is used to encrypt the ticket
- TGS is the server name in AS.REP
- Key of Alice is used to encrypt AS.REP

Unless all these conditions are met, the 4758 rejects the request. As a result, the adversary cannot fool the 4758 into generating tickets and replies for his advantage.

Ticket Granting Service

Figures 3.6 and 3.7 show ticket granting service with and without the 4758.

In the protocol using the 4758, all the encryption is done in the 4758, and no key is in the host in the clear. A consistency check similar to the one in AS takes place.

![Figure 3.6: Ticket Granting Service without the 4758](image)

As in Figure 3.4, some information is omitted.

![Figure 3.7: Ticket Granting Service with the 4758](image)

As in Figure 3.5, security critical tasks are moved to the 4758.

3.2.3 Security Discussion

In this section, I discuss the security of the design presented in Section 3.2.2.
Model

I start with constructing a model of a Kerberos realm. It consists of the following participants:

**Alice (A)** A Kerberos principal who uses authentication service and ticket granting service. Alice’s workstation is assumed to be trustworthy. This allows me to combine Alice and her workstation into one object.

**Bob (B)** A Kerberos principal with whom Alice wants to establish mutual authentication. Bob’s workstation is assumed to be trustworthy.

**KDC-host** Software component of KDC that resides on the host computer.

**KDC-4758** Software component of KDC that resides on the secure coprocessor.

**Mallory (M)** An adversary.

In this section, both the KDC and the TGS are referred as the “KDC” because they usually run on the same computer, and therefore share the same security properties.

**Threats**

I make the following assumptions in the model. Some of these assumptions are discussed in detail in Section 3.1.3.

1. System administration is appropriately done.

As problems of system administration are out of this section’s scope, administration is assumed to be done appropriately, namely, (1) the master key is stored in KDC-4758 and nowhere else, and (2) the keys of Alice and Bob are stored in
KDC-host encrypted with the master key. I discuss more about administration in Section 3.2.6.

2. Client workstations are secure.

As problems of the security of client workstations are out of this section’s scope, client workstations are assumed to be secure, namely, (1) a client workstation does not steal a user’s information, and (2) it does not alter or modify messages the user sends.

3. Keys of Alice and Bob are good.

The problem of dictionary attack against user chosen passwords is out of this section’s scope; keys of Alice and Bob are strong so that dictionary attack against them is impossible.

4. DES is strong.

My principal cipher is DES, which is assumed impossible to compromise in reasonable amount of time.

5. Mallory can compromise host.

Mallory can read and modify any information in KDC-host, and can make KDC-host do anything she wants.


Mallory can neither read nor modify any information in KDC-4758. When she tries, 4758 deletes all the information in it. Mallory cannot influence the behavior of KDC-4758.

7. Mallory can read, modify, and alter messages in the network connecting the participants.
8. Mallory can be a principal in the realm.

**Attacks**

**Key Theft**

**Without 4758**

Mallory can steal all the keys by compromising KDC-host. This is possible by Assumption 5.

**With 4758**

Mallory cannot steal any key. The master key is in KDC-4758, and is not readable (Assumption 1, 6). All the other keys are in KDC-host, but are encrypted with the master key, with DES, which is unbreakable (Assumption 4).

**User Impersonation**

**Without 4758**

Mallory can impersonate any user by stealing the user key.

**With 4758**

Mallory cannot impersonate any user. First, she cannot steal the user key. Second, the other way of impersonating a user (Alice) is to obtain a ticket, e.g., \{Alice, TGS, K_{A,TGS}\}K_{TGS} and the session key K_{A,TGS}. Mallory can obtain the ticket by sniffing the network (Assumption 7), but not the session key. The session key is generated in KDC-4758 and is always encrypted with K_A or K_{TGS} when it is outside KDC-4758. Because K_A and K_{TGS} are strong (Assumption 3), the session key cannot be obtained. These keys cannot be stolen from the client workstation (Assumption 2), either.

**Ticket / Reply Forgery (On-Line Attack)**

**Without 4758**

Mallory can generate any ticket or reply by using stolen keys.
With 4758

Mallory cannot generate a ticket or reply to her advantage. KDC-4758 generates them only after Alice shows possession of her key through preauthentication, and the consistency of the request is checked as described in Section 3.2.2. For example, when Mallory tries to make the 4758 send a ticket, created for Alice, but encrypted under Mallory's key (Mallory has her key by Assumption 8), the consistency check denies it.

3.2.4 Implementation

I implemented the AS and TGS protocols described in Section 3.2.2 by modifying Kerberos V5-1.0.6 distributed by MIT. The host platform is Linux 2.0.36 (RedHat 5.2) on an IBM PC. The secure coprocessor is the IBM 4758 Model 1, with the Secure Cryptographic Coprocessor toolkit version 1.33.

Outline

Implementation was carried out in the following three steps.

- Analysis of process_as_req() and process_tgs_req(), which implement AS and TGS to identify which portions of the functions should be moved to the 4758

- Implementation of the 4758 side functions that have functionality equivalent to the portions identified in the first step

- Modification of the host side program to make calls to the implemented functions in the 4758
Step1: Functionality Analysis

There are six parts to be moved in AS: three calls to key decryption and one each to preauthentication, TGT encryption, and \texttt{AS\_REP} encryption. Likewise, there are six parts in TGS: two calls to key decryption and one each to ticket decryption, authenticator decryption, ticket encryption, and \texttt{TGS\_REP} encryption.

As the performance evaluation in Section 3.2.5 shows, the overhead of calling the 4758 is high. Therefore, to obtain high performance, the six calls should be combined into one call. However, as cryptographic code and non-cryptographic code are tightly coupled together in Kerberos, doing so changes the order of execution and breaks modularity, thus significantly complicating the host side code. For this prototype, I decided to make six calls in each AS and TGS, valuing simplicity and manageability over performance. A detailed look at the overhead in Section 3.2.5 explains my decision.

Step2: 4758 Side Functions

Authentication Service

Key Decryption

The user keys are sent from the host to the 4758 and are decrypted. Function \texttt{decrypt\_key()} stores them in RAM. The function is called three times in AS: first for Alice's key for preauthentication, second for TGS's key for ticket encryption, and third for Alice's key for \texttt{AS\_REP} encryption. \footnote{I can save one call by caching the key in preauthentication and using it in \texttt{AS\_REP} encryption. I did not try this optimization for the prototype; performance is not yet my primary goal.}

Preauthentication

Preauthentication takes place in the following three steps:

- Alice sends to the KDC a timestamp encrypted with her key: \( \{\text{time}\}K_A \).
• KDC decrypts \{\text{time}\}K_A.

• KDC checks whether the time value is within clock skew allowed. KDC goes on
to the next step in AS if the answer is yes. Otherwise, KDC rejects the request.

Because this step requires the use of Alice’s user key, this function is moved to
the 4758, using its real-time clock for timestamp verification.

Ticket Encryption

Ticket \(\{\text{Alice, TGS, } K_{A,TGS}K_{TGS}\}\) encryption is carried out in the 4758. Some
part of a ticket is not security critical, and is generated in the host. The 4758 receives
this from the host, fills it with a session key, encrypts it with TGS’s key, and sends
it back.

AS\_REP Encryption

Similar to the ticket, \texttt{AS\_REP} \(\{\text{TGS, nonce, } K_{A,TGS}K_A\}\) includes a public part,
which is encoded in the host. The 4758 receives it, fills the session key from the ticket
encryption, encrypts it with Alice’s key.

Ticket Granting Service

As in AS, six calls are made to the 4758 in TGS. Some of the functions appear in AS;
I explain only the ones do not appear in AS.

Ticket Decryption

TGS decrypts TGT \(\{\text{Alice, TGS, } K_{A,TGS}K_{TGS}\}\) and obtain Alice’s name and the
session key. Because this step involves the TGS’s key, it must be carried out in the
4758. The 4758 decrypts the TGT, masks the session key, and returns it to the host.
The session key is stored in RAM and used in later steps.
Authenticator Decryption

The 4758 decrypts the authenticator \{Alice, time\}K_{A,TGS}, and checks the timestamp.

Step 3: Host Side Modification with Secure Hardware RPC

As with smartcards, the communication methods between the host and the 4758 are primitive. The interface is (1) host sends an array of bytes to the 4758, and (2) the 4758 returns an array of bytes. It is the developers’ responsibility to construct a high level abstraction from this interface, e.g., marshaling basic types (int, short) and more complicated data structures, dealing with endian problems, handling message buffers, and handling errors.

To improve this, I developed the Secure Hardware Remote Procedure Call (SHRPC), which provides a procedure call abstraction. The stub generator of SHRPC parses the interface definition file and generates C programs to handle the low-level communication details. With SHRPC, modification in the host side is merely to call SHRPC stub functions, e.g., decrypt_key().

I created my own simple interface definition language (IDL) for rapid implementation. I do not explain the IDL here, but the following interface definition file used in the Kerberos / 4758 integration project provides an example.

```plaintext
# Interface Declaration File
# for the Kerberos V5 / 4758 Project
# 8/6/1999, Naomaru Itoi
PROG: krb5_4758
FUNC: decrypt_key
IN:
  int type
# type :
# 0: client key
# 1: server key
string enc_key
OUT:
```

\(^9\)Communication methods for smartcards are discussed in Section 4.1.
int tick
...

The IDL should follow some standards, such as rpcgen, in the future. I will
discuss this more in Section 6.3.

The interface definition above defines a function decrypt_key() in a program
krb5_4758, an input data structure with an integer and a string of bytes (encrypted
key), and an output data structure with an integer. SHRPC generates the following:

- A header file to define the string type, the input data structure, and the output
data structure
- Utility functions, such as marshaling and unmarshaling
- Host side stub functions, which are called from the application program in the
  host
- A 4758-side template program, which receives a call from the host and dis-
  patches it to an appropriate service function in the 4758

SHRPC reduces the developer’s work to having only to write (1) the application
that calls the host side stub, and (2) the service function in the 4758.

3.2.5 Performance Evaluation

I evaluate the performance of the prototype in the following environment: IBM Netfin-
ity PC with Intel 300 MHz Pentium; the IBM 4758 secure coprocessor model 1; KDC
and Kerberos clients running on the same computer to avoid network delay. Each
measurement was carried out ten times and an average is presented in tables. Variance
was small.
Overall Result

AS is measured with kinit. The total time kinit spends with or without 4758 is shown. To exclude the time spent for password typing, the password is hard coded in the kinit program. kinit with the 4758 takes 47% more time than kinit without it. All the numbers are in milliseconds.

<table>
<thead>
<tr>
<th></th>
<th>time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>kinit without 4758</td>
<td>60.4</td>
</tr>
<tr>
<td>kinit with 4758</td>
<td>88.9</td>
</tr>
</tbody>
</table>

sclient is the TGS client I used. sclient with the 4758 takes 34% more time.

<table>
<thead>
<tr>
<th></th>
<th>time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sclient without 4758</td>
<td>71.9</td>
</tr>
<tr>
<td>sclient with 4758</td>
<td>95.3</td>
</tr>
</tbody>
</table>

In the following sections, I look into the details of the overhead introduced by the 4758 integration.

Communication Overhead

In this section, I examine the communication overhead between the host and the 4758. I measure the total time spent for the six cryptographic operations, the time spent in the 4758 (I call this 4758 time), and derive the communication overhead. As shown in Figure 3.8, the total time is the sum of the 4758 time and the communication overhead.

<table>
<thead>
<tr>
<th></th>
<th>Total (ms)</th>
<th>4758 Time (ms)</th>
<th>Communication Overhead (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS w/o 4758</td>
<td>0.54</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AS w/ 4758</td>
<td>26.92</td>
<td>9.79</td>
<td>17.13</td>
</tr>
<tr>
<td>TGS w/o 4758</td>
<td>0.32</td>
<td>-</td>
<td>17.13</td>
</tr>
<tr>
<td>TGS w/ 4758</td>
<td>27.48</td>
<td>8.67</td>
<td>18.81</td>
</tr>
</tbody>
</table>

Table 3.6: Kerberos / 4758 - Total Time, 4758 Time, Communication Overhead
Figure 3.8: Kerberos / 4758 Performance - Total Time, 4758 Time, and Communication Overhead

The result is shown in Table 3.6. Communication overhead is approximately twice as much as the 4758 time in both AS and TGS. This is an obvious bottleneck and there is an obvious optimization. Theoretically, the number of calls can be reduced from six to one in AS and two in TGS. In AS, all operations can be done at once. In TGS, the TGT must be decrypted to obtain the name of the client before the KDC looks up its database, and ticket encryption and reply encryption must happen after the database lookup. Therefore, TGS requires two calls. This optimization would reduce the overhead to 11.45 ms (19%) in AS and 14.94 ms (21%) in TGS.

4758 Time Details

Although communication overhead is the bottleneck, it is also useful to study the details of the time spent in the 4758. Breakdown of AS and TGS is shown in Table 3.7 and 3.8. For each function, the total time and the time spent in the main components are presented.

decrypt_key is invoked 3 times in AS. Three calls to decrypt_key and one call to each of the other functions sums up: $1.15 \times 3 + 1.46 + 2.53 + 2.36 = 9.80$.

decrypt_key is invoked twice in TGS. Two calls to decrypt_key and one call to each of the other functions sums up: $1.12 \times 2 + 1.32 + 1.23 + 2.11 + 1.77 = 8.67$.

DES operation takes the longest. Although the 4758 has a very fast DES hardware
<table>
<thead>
<tr>
<th>function</th>
<th>contents</th>
<th>time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>decrypt_key</td>
<td>24B DES decryption</td>
<td>0.95</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>1.15</td>
</tr>
<tr>
<td>kdc preauth</td>
<td>40B DES decryption</td>
<td>0.96</td>
</tr>
<tr>
<td></td>
<td>CGGetTime</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>1.46</td>
</tr>
<tr>
<td>encrypt tk</td>
<td>168B DES encryption</td>
<td>1.16</td>
</tr>
<tr>
<td></td>
<td>random number gen</td>
<td>0.32</td>
</tr>
<tr>
<td></td>
<td>random number gen</td>
<td>0.32</td>
</tr>
<tr>
<td></td>
<td>168B CRC</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>2.53</td>
</tr>
<tr>
<td>encode_kdc</td>
<td>216B DES encryption</td>
<td>1.24</td>
</tr>
<tr>
<td></td>
<td>random number gen</td>
<td>0.32</td>
</tr>
<tr>
<td></td>
<td>216B CRC</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>2.36</td>
</tr>
</tbody>
</table>

Table 3.7: Kerberos / 4758 Performance - Authentication Service Breakdown

<table>
<thead>
<tr>
<th>function</th>
<th>contents</th>
<th>time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>decrypt_key</td>
<td>24B DES decryption</td>
<td>0.96</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>1.12</td>
</tr>
<tr>
<td>decrypt tk</td>
<td>168B DES decryption</td>
<td>1.17</td>
</tr>
<tr>
<td></td>
<td>168B CRC</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>1.32</td>
</tr>
<tr>
<td>rd_rec dec</td>
<td>120B DES decryption</td>
<td>1.11</td>
</tr>
<tr>
<td></td>
<td>120B CRC</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>1.23</td>
</tr>
<tr>
<td>encrypt tk</td>
<td>168B DES encryption</td>
<td>1.19</td>
</tr>
<tr>
<td></td>
<td>random number gen</td>
<td>0.35</td>
</tr>
<tr>
<td></td>
<td>random number gen</td>
<td>0.35</td>
</tr>
<tr>
<td></td>
<td>168B CRC</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>2.11</td>
</tr>
<tr>
<td>encode_kdc</td>
<td>184B DES encryption</td>
<td>1.21</td>
</tr>
<tr>
<td></td>
<td>random number gen</td>
<td>0.35</td>
</tr>
<tr>
<td></td>
<td>184B CRC</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>1.77</td>
</tr>
</tbody>
</table>

Table 3.8: Kerberos / 4758 Performance - Ticket Granting Service Breakdown
engine (20 MB/s\textsuperscript{10}), DES operation takes much more time than software operations, e.g., CRC checksum. This is because a long time is spent in making a system call to the DES hardware and setting up key schedules. For applications that operate on such small data (100 - 200 bytes), which many security systems are, I suggest to provide (1) software implementation of crypto operations to save system call overhead and (2) a decoupled API for DES key scheduling separate from DES operation.

\section*{3.2.6 Discussion}

\textbf{Implementation Limitations}

Due to time limitation, my implementation has the following limitations.

\begin{itemize}
  \item User Name - Key Binding

    In Sections 3.2.2 and 3.2.3, discussions were made assuming a user name and the key of the user are encrypted together. However, this is not the case in my prototype because the key data structure in MIT Kerberos V5-1.* does not include the user name in it.

  \item Preauthentication Failure

    When preauthentication fails, either because it is not encrypted with an appropriate key, or timestamps do not match, the 4758 should reject the following operations, namely, ticket encryption and reply operations. This has not been implemented yet.

  \item Consistency Check

    Consistency check described in Section 3.2.2 has not been implemented.

  \item TGS Authenticator Check

\textsuperscript{10}50 MB/s on Model 2.
Authenticator check described in Section 3.2.4 has not been implemented yet.
The 4758 simply decrypts and returns the authenticator.

**Lessons Learned**
Integration of secure hardware into a security protocol can be significantly simplified if the original implementers of the protocol anticipate the use of secure hardware. Complication of my work comes from the cryptographic operations and the non-cryptographic operations being tightly coupled in the source code, e.g., they coexist in one function. If they are decoupled cleanly in the initial implementation, the work of integration is merely to move the crypto code to the secure hardware. Moreover, I believe the separation is good for portability of the protocol, e.g., to switch from one encryption system to another.

**Related Work**
**Public Key Based Authentication Systems**
Several public key authentication systems that are compatible with or related to Kerberos have been designed and implemented [98, 92, 96]. Many of these systems are similar to my work in that they try to protect the trusted third party. The logic to support them is that public key based authentication systems fail more gracefully than secret key based systems when the T3P is compromised. Indeed, the CA does not store private keys, thus maintaining forward secrecy and preventing an adversary from getting immediate impersonation ability. However, these systems do not lower the value of my work because of the following:

- Even in public key systems, the trusted third party (CA) is the most critical point of attack in the network. By obtaining the CA’s private key, an adversary can modify certificates, issue bogus certificates, and modify certificate revoca-
tion lists, to impersonate members. Therefore, it is vital to protect the CA with secure hardware.

- Because of both the computational overhead of public key cryptography and the necessity for key revocation, I have concerns over how public key based authentication systems scale. In contrast, the secret key based system is known to scale quite well. I believe that the secret key based Kerberos will be in service for a long time.

Therefore, I believe these systems and my work complement each other.

**Future Direction**

Several steps must be taken before this project is deployed.

**Complete Implementation**

Unfinished implementation discussed above should be completed to realize the claimed security.

**Administration**

I have not addressed problems associated with administration: changing passwords, adding / removing principals, changing the KDC’s policy, etc. These operations involve security critical keys, so they must execute in the 4758. An adversary can attack the Kerberos/4758 system by attacking the channel between the administrator and the 4758. For example, one possible attack, which could reduce the advantage of integrating the 4758 into Kerberos, is a Trojan horse in the administrator’s terminal. If it interrupts the operations by the administrator, it can steal sensitive information.

It is not possible to provide a completely trusted terminal with current commercial hardware, even with secure hardware, because they do not provide trusted I/O path.
For example, a keyboard or a display instrumented with a hardware eavesdropper can steal administrators’ keystrokes. However, it is easier to keep a terminal secure during administration than to keep a Kerberos server secure in the 24x7 fashion. Therefore, I defer solving this problem of secure I/O.

**Performance Optimization**

As described in Section 3.2.5, the six calls to the 4758 in AS and TGS should be combined into one and two calls respectively to optimize the performance. The drawback of this optimization is that it changes the Kerberos code significantly. In the Kerberos/Cartel meeting in July of 1999, I sensed that such a radical change would pose a major challenge to Kerberos developers with regard to maintaining the source code. Therefore, I decided to first implement a prototype to determine what the computer systems community thinks about it before proceeding to the deployment step.

**Brute Force Attack to Master Key**

If an adversary has access to messages passed between the host and the 4758, he or she can obtain a plaintext-ciphertext pair. Some messages are encrypted with single DES and the master key. This is problematic because given a plaintext-ciphertext pair, DES key can be cracked by a brute force attack [30]. Kerberos distribution from MIT supports triple DES, eliminating this threat.

**Replay Attack**

An adversary can use a replay attack to impersonate Alice if he or she hijacks the host and has Alice’s obsolete password. Here I describe the possible attack and the countermeasure. The Kerberos/4758 protocol protects the keys from Mallory even if she compromises the host. However, without additional measures, this protocol suffers from replay attack if Mallory learns one of Alice’s old passwords (or keys).
The replay attack works as follows:

- Mallory obtains an old key of Alice, $K_A$.

- Because Mallory has complete access to the host, she can obtain $\{\text{Alice, } K_A\}K_M$. ($K_M$ is the KDC master key.)

- Alice, knowing that her key is stolen, changes her key to $K_A'$. At this point, the old key is obsolete.

- Mallory hijacks the KDC, and makes it send $\{\text{Alice, } K_A\}K_M$ to the 4758. Since the 4758 does not know $K_A$ is obsolete, it thinks $K_A$ is good, and sends a reply encrypted by $K_A$ to the KDC/Mallory. Mallory successfully decrypts the reply, impersonates Alice.

To avoid this attack, I use key version numbering and obsolete key caching. First, all the keys in the Kerberos database have a key version number, $N$. An encrypted key entry contains this version number, i.e., $\{\text{Alice, } K_A, N\}$. When Alice changes her key, Alice’s current key version number is updated to $N+1$. The 4758 generates a new key entry $\{\text{Alice, } K_A', N+1\}$, sends the entry back to the host, and caches a pair $\{\text{Alice, } N+1\}$ in its internal memory. The 4758 checks the cache whenever it receives a key from the host. If the version numbers do not match, then the key received is obsolete. To avoid cache overflow, once in a while (e.g., daily) the 4758 regenerates the new $N$ and computes the new entries for all the keys, and sends them back to the host.

The cache should not overflow too quickly. If the cache size is 1MB and each entry is 32 bytes, then the maximum number of entries in the cache is 32K entries — which I imagine exceeds the maximum number of password changes in a day.
3.2.7 Section Summary

I have integrated the IBM 4758 secure coprocessor into Kerberos KDC. The resulting KDC will survive a usually unavoidable threat: host compromise. The following two aspects highlight the value of this work.

- Improvement to important software

Along with the Kerberos/smartcard integration in Section 3.1, this work improves Kerberos, which is an important authentication system. Compromise of KDC is a serious security threat to Kerberos, and this work solves this problem.

- Practical application for secure coprocessor

Like smartcards, secure coprocessors are application hungry. Most of the current applications do not take advantage of their tamper resistance. I have provided a practical application, and have shown that secure coprocessors can enhance the security of current security infrastructures.

This idea can be applied to other trusted third party mechanisms, e.g., CAs.
3.3 Internet Extension to Smartcard Kerberos Client

3.3.1 Introduction

Although smartcards are becoming popular, the current technology limits smartcards to local use, namely, they can only be accessed through serial ports from the workstations they are attached to. Smartcard applications, as in authentication (Section 3.1, [4]) and payment systems [28, 67], generally assume that the smartcard and the user are on the same workstation. The fact that smartcards cannot be used remotely poses a severe limitation as the following scenarios illustrate. First, consider a typical office, in which a user has several workstations providing diverse services. To receive smartcard service from all the workstations, the user is forced to install smartcard readers on all computers and move the card around as the tasks demand. Second, consider a scenario in which the user logs into a remote workstation that is physically out of her reach, and wants to use a local smartcard. She simply cannot do so without a remote smartcard access mechanism.

A solution to this problem has been developed in PC/SC-Lite [20] framework. It is among the more sophisticated card managers, in that it allows applications to access card readers on remote hosts. However, there are a few problems with 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, PC/SC-Lite does not provide a location independent name for a smartcard: a smartcard is identified by the host’s domain name and a serial port on that host. If the card moves from one host to another or if the reader is moved to a different port, the smartcard’s name becomes invalid.

To solve these problems, I have developed a smartcard remote access mechanism which provides a location independent name, and protects the information transmit-
ted. Our work pursues the approach developed by Guthery et al. [38] and Rees et al. [86], which use Internet protocol for smartcards. I adapted the smartcard web server developed at CITI [86] by adding UDP support and a protocol for secure, authenticated remote communication.

In this section, I describe the remote smartcard access system I developed. The building blocks of the 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) [51] is used to establish a session key between the smartcard and the remote workstation. Subsequent communication is encrypted with the session key.
- Kerberos and SSH clients, which have previously been modified to accommodate smartcards (Section 3.1), are further modified to take advantage of this system for remote smartcard access.

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

3.3.2 Design

Here I describe the goals of the system and the decisions I made to achieve these goals.

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.

I assign a unique, durable, Internet domain name [74] to each card. The Internet domain name service (DNS) maps the domain name to an IP address. This assumes deployment of a secure, dynamic DNS [100] or mobile IP [81].

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.

**Transport layer**

Because an IP stack for Java Card has been developed already in our lab [86], it is natural for me to choose UDP or TCP for data transmission. TCP has many advantages over UDP, namely: reliability and error correction. Nonetheless, I 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 limited resources 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.

**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 the smartcard and the host is the Internet, which is generally an insecure medium.
Consequently, my 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 [24]. My system achieves the first two properties by employing SPEKE, a secure key exchange protocol [51]. SPEKE establishes a session key for channel encryption while at the same time authenticating both parties with a shared secret.

I did not implement cryptographically secure integrity checking, or even UDP checksum in this implementation; at this time I find checksum calculation to be too time consuming for the applications. My 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.

**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 [25]. However, it does not achieve authenticity, and is vulnerable to a man-in-the-middle attack [89].

- **Encrypted Key Exchange**

  Encrypted Key Exchange (EKE) [7] 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 offer us permission to use the protocol.

- **Open Key Exchange**

  Open Key Exchange (OKE) [70] 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 I use.
$S$ 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 $Q_A = f(S)^{R_A} \mod p$  \hspace{1cm} $A \rightarrow B: Q_A$

**Step 2.** Bob computes $Q_B = f(S)^{R_B} \mod p$  \hspace{1cm} $B \rightarrow A: Q_B$

**Step 3.** Alice computes $K = h(Q_A^{R_B} \mod p)$

**Step 4.** Bob computes $K = h(Q_A^{R_B} \mod p)$

**Verification (optional)**

**Step 5.** Alice picks random number $C_A$  \hspace{1cm} $A \rightarrow B: E_K(C_A)$

**Step 6.** Bob picks random number $C_B$  \hspace{1cm} $B \rightarrow A: E_K(C_A, C_B)$

**Step 7.** Alice verifies $C_A$  \hspace{1cm} $A \rightarrow B: E_K(C_B)$

**Step 8.** Bob verifies $C_B$

Figure 3.9: SPEKE protocol
SPEKE protocol

I 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 me 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.

Figure 3.9 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. Kerberos and SSH are self-authenticating, so I omit consideration of this step in the remainder of this dissertation.

Figure 3.10 illustrates the design. $f(S)$ is precomputed and stored on the card. The host and the smartcard exchange two request/reply pairs, initiated by a connec-
tion request from the host. This signals the smartcard to generate its first message while a user is entering her PIN, possibly achieving some overlap.

Host  \[\rightarrow 1\text{ conn req}\] \quad \text{Smartcard}

\begin{align*}
\text{Compute QA} & \quad \text{Compute QB} \\
\quad \text{1' QB} & \quad \text{Send QB} \\
\text{Send QA} & \quad \text{Compute session key} \\
\text{Compute session key} & \quad \text{2' conn ok}
\end{align*}

Figure 3.10: SPEKE implemented with smartcard

### 3.3.3 Security Discussion

In this section, I consider three potential vulnerabilities that could compromise the security of the developed system: host compromise, off-line dictionary attack, and on-line attack.

**Host Compromise**

Host compromise may yield the PIN entered by a user, by either finding it in memory, or obtaining it through a Trojan horse. The lack of trusted path from the keyboard to applications on most workstations make PIN / password typing dangerous. The adversary, with the PIN in hand, can convince the smartcard that he is the legitimate user, and receive services from the smartcard.

However, losing the PIN is not as bad as losing the password. In a smartcard protected system, the adversary must obtain the PIN and must access the smartcard to impersonate the user. In contrast, in a password protected system, the adversary
can impersonate the user by obtaining only the password.

In addition, one can enhance the security of the system by taking advantage of smartcard properties: physical isolation and trustworthiness. First, if an output device is installed to indicate transaction between the host and the smartcard reader, the user is notified that the smartcard is being used when it should not be. Second, because a smartcard is a trustworthy device, one can implement a trustworthy logging system in the smartcard that reports fraudulent use.

**Off-Line Dictionary Attack**

An off-line dictionary attack does not pose a critical threat to my system. The PIN is raised to a random number, which is unknown to the adversary, when it travels through a network. The adversary cannot distinguish the right PIN from the wrong ones.

Jablon suggests a number of methods that can be used to prevent information leakage in SPEKE (and EKE) [51]; although I have not implemented all of these, I would find it prudent to do so before fully deploying the system. Following Jablon’s suggestions, I feel confident that SPEKE can effectively blunt this attack.

**On-Line Attack**

An on-line attack is a potent threat. This attack would proceed as follows. An adversary tries each candidate PIN to establish a secure channel with the smartcard. 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.

If four-digit PINs are used (which users may prefer because they are often used in banking cards), this attack is feasible. If I assume that a session can be completed
in five seconds, then the entire space of potential PINs can be tested by an on-line adversary in 50,000 seconds, less than a day if the card is kept online.

The best way of solving this problem is to always pull off the card from the reader after using it. However, the user may want to leave a smartcard in her office and access it remotely. In that case, she cannot pull it off the reader.

To solve this problem, I suggest a counter on the card that keeps track of failed attempts to complete a Kerberos or SSH session. When the counter reaches a certain limit (e.g., 5 failures), the card blocks itself. However, this presents a denial-of-service attack, in which the adversary tries random PINs, quickly blocking the card. I am 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 [42, 43] 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.

3.3.4 Implementation

Overview

Figure 3.11 illustrates the overview of the system. “Application” is a Kerberos or SSH client in this implementation. I moved key management and cryptographic functionality to a remote smartcard.

The Kerberos client uses the remote smartcard to carry out Authentication Service, in the way described in Section 3.1.2. 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 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 are transmitted on UDP/IP.

The 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 APDUs.

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.

I 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. The (public) modulus is a safe prime that is
#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 logical 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.

data is the data to be transmitted

Figure 3.12: SPEKE Data Structure

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 Solaris 2.6, and has been ported to Linux
2.2. The tunneling host runs on OpenBSD 2.7. 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.

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 the smartcard user. Although I have implemented Kerberos and SSH, I describe only the Kerberos implementation in the remainder of this section, as 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 is similar to the one developed in Section 3.1, except that it uses the SPEKE library to communicate with a smartcard on a remote host instead of communicating to a locally-attached reader.

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 AS_REP 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 3.12.

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 [109], the GNU Multiple Precision Arithmetic Library [34], and the CTC library [73].

**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 attach and strip IP headers in APDUs. The routing table of the smartcard’s host is configured so that the tunnel daemon receives packets directed to the smartcard.

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 the sc7816 library. When 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 master key; the SSH application signs a random challenge with a private RSA key.

Thanks to the object oriented style of programming supported by Java Card 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:

```java
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);
        }
    }
}
```

84
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 inspects the header of a packet. If the packet is a special message for key exchange (e.g., connection request), this method processes this, creates an appropriate return 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.

```java
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.
3.3.5 Performance Evaluation

Here I discuss the performance of this system. This system is not fast. Therefore, I focus on highlighting system bottlenecks, and discuss how performance can be improved.

Performance evaluation was carried out on two workstations on a 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 38.4 Kbps.

Execution time

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

<table>
<thead>
<tr>
<th></th>
<th>remote</th>
<th>local</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kerberos</td>
<td>12.8</td>
<td>3.33</td>
</tr>
<tr>
<td>SSH</td>
<td>12.6</td>
<td>3.43</td>
</tr>
</tbody>
</table>

The remote versions are much slower than the local ones.\(^\text{11}\) 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.

Details

Here I discuss the execution time of the Kerberos client. These observations also apply to the SSH client.

11 The number for local Kerberos differs from the result reported in Section 3.1, as the MIT KDC (used in this section) generates a larger AS.REP than the Heimdal KDC (used in Section 3.1), and kinit in this section does not save the TGS session key in the smartcard.
are reported in seconds and are the average of five time trials. Variance is negligible.

<table>
<thead>
<tr>
<th>time (s)</th>
<th>events</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>kinit start</td>
</tr>
<tr>
<td>0.02</td>
<td>SPEKE connect start</td>
</tr>
<tr>
<td>0.03</td>
<td>Host send SPEKE1 (connect request)</td>
</tr>
<tr>
<td>0.03</td>
<td>Host send SPEKE2 (Q_A)</td>
</tr>
<tr>
<td>2.07</td>
<td>Host recv SPEKE1 (Q_B)</td>
</tr>
<tr>
<td>3.56</td>
<td>Host recv SPEKE2 (connect ok)</td>
</tr>
<tr>
<td>3.56</td>
<td>get_key_table start</td>
</tr>
<tr>
<td>5.88</td>
<td>get_key_table finish</td>
</tr>
<tr>
<td>5.88</td>
<td>decrypt ticket start</td>
</tr>
<tr>
<td>9.93</td>
<td>decrypt ticket finish</td>
</tr>
<tr>
<td>9.93</td>
<td>decrypt ticket finish</td>
</tr>
<tr>
<td>12.80</td>
<td>decrypt ticket finish</td>
</tr>
<tr>
<td>12.80</td>
<td>kinit end</td>
</tr>
</tbody>
</table>

Table 3.9: Kerberos on SPEKE Timeline

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, as shown in Table 3.10

<table>
<thead>
<tr>
<th>request type</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPEKE1 (→ connect request, ← Q_B)</td>
<td>2.04</td>
</tr>
<tr>
<td>SPEKE2 (→ Q_A, ← connect ok)</td>
<td>1.49</td>
</tr>
<tr>
<td>get_key_table request (← princ table)</td>
<td>2.33</td>
</tr>
<tr>
<td>decrypt block 1 (→ data, ← data)</td>
<td>4.06</td>
</tr>
<tr>
<td>decrypt block 2 (→ data, ← data)</td>
<td>2.87</td>
</tr>
</tbody>
</table>

Table 3.10: Kerberos on SPEKE Breakdown

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

- time spent in the smartcard
\[ S \] 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 3.13: EKE Protocol
The optional verification step is not shown.

- IP communication between user’s host and smartcard’s host
- overhead of the tunnel daemon
- sc7816 library overhead
- communication between smartcard and smartcard’s host. (This includes time for \texttt{get-response} APDU.)

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

<table>
<thead>
<tr>
<th>event</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP + tunnel + sc7816</td>
<td>0.00</td>
</tr>
<tr>
<td>in-the-card</td>
<td>1.83</td>
</tr>
<tr>
<td>card communication</td>
<td>0.21</td>
</tr>
<tr>
<td>Total</td>
<td>2.04</td>
</tr>
</tbody>
</table>

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 my 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 I use do not support T=1.

**EKE measurement**

Although I cannot use it in my projects because of a patent issue, I implemented and measured EKE to satisfy my curiosity. EKE is a simple and well-known protocol. The EKE protocol, described in Figure 3.13, is implemented with one pair of messages and optional verification. Like my SPEKE implementation, I 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 in Table 3.11. EKE takes 4.47 seconds to complete connection establishment, compared to 3.56 seconds for SPEKE.

<table>
<thead>
<tr>
<th>time (s)</th>
<th>events</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>EKE connect start</td>
</tr>
<tr>
<td>0.01</td>
<td>Host send EKE1 (connect request)</td>
</tr>
<tr>
<td>1.43</td>
<td>Host send EKE2 ($E_S(P)$)</td>
</tr>
<tr>
<td>4.45</td>
<td>Host recv EKE2 ($E_S(E_P(K))$)</td>
</tr>
<tr>
<td>4.47</td>
<td>EKE connect ok</td>
</tr>
</tbody>
</table>

Table 3.11: EKE Timeline

Time taken for each message pair is shown in Table 3.12. 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, I expect key generation time to improve with new
generations of microprocessors.

<table>
<thead>
<tr>
<th>request type</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EKE1 (→ connect request, ← NULL)</td>
<td>0.83</td>
</tr>
<tr>
<td>EKE2 (→ $E_{S}(P)$, ← $E_{S}(E_{P}(K))$)</td>
<td>2.95</td>
</tr>
</tbody>
</table>

Table 3.12: EKE Breakdown

### 3.3.6 Section Summary

I designed and implemented an Internet-standards compliant middleware infrastruc-
ture 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, I find the infrastructure useful, and anticipate that it will
enable many new types of smartcard applications.

The following four aspects highlight the value of this work.

- **Useful and necessary.**

  I implemented and deployed the sc7816 version of the smartcard-integrated
  Kerberos described in Section 3.1 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 con-
nected to the Internet. This work is the first implementation of such a system.

- **Standard API.**
The developed protocols are built on UDP and IP, universally accepted communication standards. I 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.

The system enables developers to implement IP-based smartcard applications easily. The source code is freely available on the CITI smartcard web page.
Chapter 4

Secure Hardware Integration with Secure Storage

Secure storage, which protects data from illegal access by an adversary, is one of the most important security applications. Its importance is evidenced in the hardware theft incidents I discussed in Section 1.2.1. Although there are two approaches already established to secure storage, i.e., (1) store secrets in secure hardware, and (2) store encrypted data on a workstation, they have shortcomings. The first approach suffers from the lack of convenient access interface and the size limitation of secure hardware. The second suffers from the two problems I repeatedly mention: trusted computing and passwords.

In this chapter, I attack these problems by developing two projects:

- **Smartcard Filesystem, or SCFS**

  SCFS provides transparent access to data on a smartcard from a file system on a UNIX workstation. SCFS allows unmodified applications to take advantage of a smartcard.

- **Smartcard Secured Cryptographic File System, or SC-CFS**

  SC-CFS is a file system that encrypts files on a workstation. It counters dictio-
nary attack, and provides fine-grain key management. SC-CFS allows unmodified applications to take advantage of the file protection that is hardened with a smartcard.

4.1 Smartcard Filesystem

4.1.1 Introduction

As discussed in Section 2.3, secure hardware is suitable to store secrets because they protect information well. Smartcards are especially suited for personal secure storage, as they can be carried by users. However, they have not been widely used in this way because they are not easy to access; developers suffer from the lack of user friendly, standard communication protocol between host-side software and a smartcard. The ISO-7816 communication protocol [50] is so widely accepted that virtually all smartcards support it\(^1\). However, this protocol is not a particularly desirable one:

- It is a primitive message passing protocol. Providing only read and write operations for raw data, it does not define higher interfaces such as UNIX files and I/O streams. This hampers developers’ ability to build application software.

- Although almost all smartcards support ISO-7816, details of the protocol implementation differ among vendors and types of smartcards. This requires a software developer to tailor their applications to specific smartcards.

Differences among smartcards range from trivial ones, such as different opcodes, to essential ones, such as different authentication mechanisms, different timing requirements, etc. For example, the CLA byte of the application class\(^2\) is 0x00

---

\(^1\) Almost all smartcards support ISO-7816-1, 2, and 3. Many support ISO-7816-4 [88]

\(^2\) CLA byte is the first byte of an APDU. For more details, please see Smart Card Developer’s Kit book [39] or ISO-7816 standardization [50]
in some smartcards (Giesecke & Devrient STARCOS Version 2.1), while it is 0xc0 in others (Schlumberger MultiFlex).

To address the problems of ISO-7816, many new standards have been proposed. The examples are:

- General purpose standards: Open Card Framework (OCF) [21, 47] and PC/SC [27, 22]
- Special purpose standards: PKCS #11 [62] for cryptography, EMV [28] and SET for electronic commerce

Although these standards provide higher level abstractions than ISO-7816-4, it remains a challenging task for developers to choose a standard, purchase all software and hardware required, learn API and tools, and finally implement software. Furthermore, those standards do not eliminate problems with interoperability, e.g., PC/SC is used only with Windows operating system. They create their own API dependencies because software written for one standard does not run with another. I discuss these issues more in Section 4.1.5.

My solution to this problem is to embrace a classical, sophisticated API - the UNIX filesystem - instead of inventing a new one. The UNIX filesystem abstraction suits smartcards well when they are used for secure storage. In addition, it allows developers to use sophisticated UNIX file access commands, such as $ls$, cat, cd, etc.

In UNIX operating systems that support a Virtual Filesystem [55, 71], it is possible to write a file system that communicates with a special hardware device (e.g. a smartcard) and mount it in the filesystem. The mounted special hardware device is identical to the normal UNIX files from the perspective of a user or application software. For example, if a smartcard is mounted on /smartcard, it is possible to
use UNIX commands such as \texttt{ls}, \texttt{cd}, \texttt{pwd}, and \texttt{cat}, and system calls such as \texttt{open}, \texttt{read}, and \texttt{write} to files in the smartcard.

I have implemented Smartcard Filesystem (or SCFS) in OpenBSD-2.2\textsuperscript{3}, and later ported it to OpenBSD-2.4 and 2.6. SCFS is loaded into an OpenBSD kernel. With SCFS mounted, a user or an application can use files in a smartcard as normal UNIX files.

This section presents the design, the implementation, and the performance of SCFS. Performance evaluation in Section 4.1.4 shows that overhead of SCFS is small (under 2.4\% of total access time) so that it does not degrade the performance of smartcard software. I compare SCFS with other standards in Section 4.1.5.

4.1.2 Design

Design Goals

To provide a user friendly interface to a smartcard, I set the following design goals. Some of them are not achieved, and the reasons are discussed in Section 4.1.2:

- Files in a smartcard should be indistinguishable from other UNIX files.

- A smartcard can be accessed with UNIX file access system calls (e.g., \texttt{creat}, \texttt{open}, \texttt{read}, and \texttt{write}).

- UNIX commands (e.g., \texttt{ls}, \texttt{cd}, \texttt{pwd}, and \texttt{cat}) can be used to access files in a smartcard.

- SCFS should be able to access any smartcard that supports ISO-7816.

- SCFS should hide details about a smartcard to users.

- Security of a smartcard must be preserved.

\textsuperscript{3}OpenBSD is a free, 4.4BSD-based operating system. http://www.openbsd.org
• No smartcard files may be cached in the host because a smartcard is a more secure place to store data (see the end of Section 4.1.2).

Design Problems

A huge obstacle to achieving the goals is the lack of metadata in a smartcard. Some information essential for the UNIX filesystem is simply not present in a smartcard, e.g., logical file sizes, directory contents, and time stamps. Without such information, it is impossible to implement the complete functionality of the UNIX filesystem. For example, without directory entries, it is impossible to implement ls properly. I have three choices to counter this problem, with tradeoffs:

• Dictate an internal format on a smartcard to store information such as directory entries and length of a file. This provides full functionality of UNIX filesystems. However, smartcards without this information cannot be accessed by SCFS.

• Store the metadata on the host computer. This provides full functionality of UNIX filesystems, and can be done without dictating internal format on smartcards. However, this forces every host to know about every smartcard, makes SCFS less configurable and less scalable.

• Degrade functionality of SCFS. For example, no ls, no cat.

I compromise by storing essential, but minimal metadata on a smartcard. I believe it is essential to be able to determine a smartcard’s directory structure through UNIX commands such as ls, so SCFS requires directory structure information to be stored in a smartcard. I also require a smartcard to store file lengths because they are necessary to implement the read and write system calls. Every directory (or DF in

96
ISO-7816 terminology) in a smartcard has a file called 2e.69 (“i”4) containing the requisite metadata.

**Design**

Inspired by Arla [104], SCFS has a two-layer structure: one is the kernel module, *xfs*, which handles VFS requests, and the other is the user daemon, *scfsd*, which communicates with a smartcard. Figure 4.1 shows the overview of the design.

![Figure 4.1: SCFS Design](image)

When an application calls a VFS operation (e.g., `read` or `write` to a smartcard file), the kernel module upcalls *scfsd* to request service. *scfsd* creates APDUs (e.g., to read a file), and sends them to a smartcard, gets returned data, and passes it to the kernel module.

Separation between *xfs* and *scfsd* allows us to use an existing ISO-7816 library [85] for handling the protocol and dealing with its complex timing requirements. Kernel code is minimized, making SCFS easy to debug and port.

To absorb differences among smartcards, SCFS requires some information about a smartcard before it is mounted, e.g., existence of special APDUs, CLA bytes, *Answer To Reset (or ATR)* they return, etc. The information is stored in a SCFS

---

\(^4\)“i” for “inode”. 

---

97
configuration file, /etc/scfs.scdb by default.

SCFS automatically identifies a smartcard type from its ATR. When a reset signal is sent to a smartcard, it responds with a 4 - 32 byte ATR, unique to each smartcard type. The SCFS configuration file has a database of known ATRs. If the ATR from the smartcard is listed in the configuration file, SCFS retrieves the entry for that type of smartcard. Details about the configuration file are described in Section 4.1.3.

Unlike most UNIX filesystems, SCFS does not cache data because caching might degrade the security of data resident in a smartcard. Data in the UNIX filesystem (typically a hard disk) is not protected as securely as in a smartcard.

4.1.3 Implementation

Overview

As described in Section 4.1.2, SCFS is separated into two parts. The kernel module (xfs), the SCFS daemon (scfsd), and the communication mechanism between them are detailed in this section. Implementation of SCFS is based on Arla-0.19.

Kernel Module (xfs)

The kernel module (xfs) implements a virtual filesystem, the ioct1 system call, and communication with scfsd. It consists of several functions called by the kernel when a file in SCFS is accessed. For example, the read system call is handled by the xfs_read().

When xfs needs to communicate with a smartcard, it upcalls scfsd. For example, xfs_read() invokes xfs_message_readsc() in scfsd. xfs waits until it receives data from scfsd, and sends the data back to the application with the uiomove kernel function.
SCFS daemon (scfsd)

scfsd performs operations requested by xfs. For requests that require smartcard access, scfsd translates the request to ISO-7816 APDUs. Figure 4.2 shows an example of message flow when an application requests to read 8 data bytes from a smartcard.

![Diagram showing message flow](image)

Figure 4.2: Reading 8 Byte Data from Smartcard

Communication between xfs and scfsd

xfs communicates with scfsd through RPC. It constructs a request message, puts it into a message queue, and waits for scfsd to reply. Code for sending a request to read 8 bytes from a smartcard is as follows:

```c
struct xfs_message_readsc msg;

msg.header.opcode = XFS_MSG_READSC;
msg.buf = buf;
msg.size = 8;
msg.offset = 0;
fidcpy (msg.fid, xmode->handle);
xf s_message_rpc(fd, &msg.header, sizeof(msg));
```
After invoking `xfs_message_rpc()`, `xfs` sleeps until it receives the reply. `scfsd` receives data from a smartcard and sends it back to `xfs`. Here is an example of sending a reply message:

```c
struct xfs_message_installdata msg;

msg.header.opcode = XFS_MSG_INSTALLDATA;
memcpy (msg.buf, data);
msg.size = size;
xs_send_message_wakeup(fd, error, msg);
```

**Important VFS/Vnode operations**

In this section, I detail the implementation of some important VFS and vnode operations.

VFS Operations:

- **`xfs_mount()`** mounts SCFS.

  It first sends a reset signal to the smartcard. When it receives an ATR, it scans the configuration file to find the entry with the matched ATR, reads the configuration information, initializes `scfsd` and `xfs`, and creates the mount point.

- **`xfs_root()`** operation selects a root directory (.3f.00) in a smartcard and installs an XFS node and a `vnode` for a root node.

Vnode operations:

- **`xfs_lookup()`** translates a path to an 8 byte *file ID, or fid.*
A **fid** is a file identifier that is unique in SCFS, consisting of names of the file itself and its ancestors. For example, a **fid** of a file 3f.00/77.77/77.01 is 77.01.77.77.3f.00.ff.ff.

It checks if the requested pathname and its parent are both in the directory structure. If they are, it constructs and returns the **fid**. Currently, a path length is restricted to four components because a **fid** is 8 bytes long, big enough to hold four ISO-7816 file names, which are two bytes each. We map these two bytes into their ASCII equivalents in the natural way.

- **xfs_read()** reads data from a (possibly PIN-protected) smartcard file.

Its operation are as follows.

1. It selects the target file. When the current file and the target file have the same parent, the target file is selected by a select APDU. Otherwise, the entire path from the root must be navigated; ISO-7816 does not allow selection of an arbitrary file, only one in the currently selected directory, so in this case, **xfs_read()** selects the root file (3f.00), and moves down a path one by one to the target file.

2. Send read APDU (e.g., c0 b0 00 00 length) to the smartcard.

3. If the read fails because the file is protected by a PIN, prompt the user for a PIN. The prompt is directed to the controlling tty of the application that issued the system call. Get the PIN, and send it to the smartcard. If this succeeds, try read again.

4. Pass the data back to the user via XFS and by **uiomove()**.

- **xfs_write()** behaves identically to **xfs_read()**, except for the direction of data.
• `xfsgetattr()` installs a VFS attribute structure (`struct vattr`) and an XFS attribute structure (`struct xfs_attr`).

`scfsd` performs the actual construction of the XFS attribute structure and sends it to `xfs`, which converts it into a VFS attribute structure.

• `xfs.readdir()` constructs directory entries.

This is typically called by a `getdirent()` system call, often as a result of an `ls` command. It returns directory entries (`struct dirent`) of a selected directory. Each entry describes a file or a directory in the selected directory. ISO-7816 shortcomings require me to define my own metadata strategy, described in Section 4.1.2. `xfs.readdir()` constructs full directory entries from the directory entries and from the metadata file and returns them to the application.

Some functionalities in a smartcard do not fit in the concept of a filesystem. For example, there is no system call to read a PIN to authorize a user. However, these functionalities are necessary to take advantage of security features of a smartcard. To incorporate them into SCFS, I use the `ioctl()` operation.\(^5\) `ioctl()` takes an opcode and data and performs an opcode-specific action.

Implementation of `ioctl()` is straightforward, translating one opcode to one APDU. `ioctl()` implements create file, verify PIN, verify key, internal authentication, external authentication, get response, and get challenge APDUs.

**Configuration File**

The configuration file (`/etc/scfs.scdb` by default) includes entries for ATR, the name of the smartcard, the CLA byte used for APDUs, whether some APDUs are supported, the type of supported PIN protection, etc. An example of a configuration file is as follows:

\(^5\)I use `ioctl()` to avoid adding a new system call; this decision should be revisited someday.
ATR 3b 32 15 0 49 10 {
  CARDNAME         CyberFlex
  MULTIFLEXPIN     no
  MULTIFLEXGETRES  no
  CLA_DEFAULT      c0
  CLA_VERIFYKEY    f0
  CLA_READBINARY   f0
  CLA_UPDATEBINARY f0
  CLA_READRECORD   -1
  CLA_UPDATERECORD -1
}

ATR 3b 2 14 50 {
  CARDNAME         MultiFlex
  MULTIFLEXPIN     yes
  MULTIFLEXGETRES  yes
  CLA_DEFAULT      c0
  CLA_VERIFYKEY    f0
}

ATR 3b 23 0 35 11 80 {
  CARDNAME         PayFlex/MCard
  MULTIFLEXPIN     no
  MULTIFLEXGETRES  no
  CLA_DEFAULT      00
}

The byte string after the “ATR” tag is matched with the ATR returned from a
smartcard at reset. The CLA_* tags defines CLA bytes for specific APDUs, used
by scfsd to construct APDUs. -1 means that the APDU is not supported in the
smartcard type. If a CLA byte is not specified for the APDU, CLA_DEFAULT
is used. For example, in CyberFlex, the CLA byte is 0xf0 for the verify_key,
read_binary, and update_binary APDUs. read_record and update_record APDUs
are not defined. 0xc0 is used for the CLA byte for the other APDUs.
4.1.4 Performance Evaluation

Here I evaluate the performance of SCFS, tested on two Schlumberger cards, MultiFlex and CyberFlex Access. Our test harness is based on a 400 MHz Pentium running OpenBSD-2.4.

Serial communication with smartcards uses 12 bits per byte (one start, eight data, one parity, two stop bits). Our test harness communicates with smartcards at 38.488 Kbps, or 312 μsec. per byte.

SCFS Overhead

First, I evaluate the overhead introduced by SCFS. I measure the total elapsed time and the smartcard access time for read and write operations. The difference reflects the filesystem overhead. Figure 4.3 shows this relation.

![](image)

Figure 4.3: Total Time, Smartcard Time, and SCFS Overhead

The result of the measurement with Multiflex is shown in Table 4.1. Command names, the total time, the smartcard access time, and the SCFS overhead are shown. Numbers are in milliseconds. The result shows that the SCFS overhead is up to 2.4% of the total time, i.e., negligible.
<table>
<thead>
<tr>
<th>Command</th>
<th>Total (ms)</th>
<th>Smartcard (ms)</th>
<th>Overhead (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read 8 bytes</td>
<td>28.9</td>
<td>28.2</td>
<td>0.7 (2.4%)</td>
</tr>
<tr>
<td>Read 128 bytes</td>
<td>190.2</td>
<td>189.4</td>
<td>0.8 (0.4%)</td>
</tr>
<tr>
<td>Write 8 bytes</td>
<td>63.4</td>
<td>62.7</td>
<td>0.7 (1.1%)</td>
</tr>
<tr>
<td>Write 128 bytes</td>
<td>1259.5</td>
<td>1258.9</td>
<td>0.7 (0.1%)</td>
</tr>
</tbody>
</table>

Table 4.1: SCFS Overhead for Read and Write

**Other Operations**

Second, I evaluate the SCFS overhead for other operations. Table 4.2 shows the cost of them on CyberFlex Access.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Card Time (msec)</th>
<th>SCFS Overhead (msec)</th>
<th># Bytes Transferred</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>0.00</td>
<td>0.57</td>
<td>0</td>
</tr>
<tr>
<td>lseek</td>
<td>0.00</td>
<td>0.34</td>
<td>0</td>
</tr>
<tr>
<td>create</td>
<td>466.00</td>
<td>2.00</td>
<td>78</td>
</tr>
<tr>
<td>remove</td>
<td>626.00</td>
<td>37.40</td>
<td>64</td>
</tr>
<tr>
<td>verifykey</td>
<td>258</td>
<td>1.67</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 4.2: SCFS Overhead for Other Operations

The effect of directory structure cache, created at mount time by reading the “.i” files, is evident in the open and lseek operations, which do not communicate with the card. SCFS does not introduce significant overhead in any of these operations.

**SCFS Read and Write Performance**

Third, I measure the read and write performance of SCFS. I used six different operand sizes, ranging from 1 byte to 254 bytes. Figure 4.4 shows the result. For both cards, elapsed time as a function of operand size is very close to linear.

Table 4.3 shows the card-related per-byte cost for these functions. The write times are substantially longer than the read times, reflecting the property of EEPROM.
Figure 4.4: SCFS Read and Write Performance

<table>
<thead>
<tr>
<th>Card Type</th>
<th>Operation</th>
<th>time (msec) per byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiflex</td>
<td>read</td>
<td>0.35</td>
</tr>
<tr>
<td>Cyberflex Access</td>
<td>read</td>
<td>0.26</td>
</tr>
<tr>
<td>Multiflex</td>
<td>write</td>
<td>5.62</td>
</tr>
<tr>
<td>Cyberflex Access</td>
<td>write</td>
<td>4.62</td>
</tr>
</tbody>
</table>

Table 4.3: SCFS Read and Write Performance
4.1.5 Discussion

Related Work

Here I discuss three important related works, OCF, PC/SC, and some special purpose standards.

OCF

OpenCard Framework is middleware that supports a smartcard with Java [21, 47] by providing high-level APIs, reader type transparency, card type transparency, and extensibility. These objectives are similar to mine. The principal advantage of OCF is that it employs Java. Programmers familiar with Java can start smartcard programming easily. The following is an example taken from “OpenCard Framework 1.1 Programmer’s Guide” [46]. It reads a file “id” (0x6964) and prints it out to standard output.

public static void main(String[] args)
{
    System.out.println(
        "reading smartcard file...");

    try {
        SmartCard.start();

        // wait for a smartcard with file
        // access support
        CardRequest cr =
            new CardRequest(
                FileAccessCardService.class);

        // read file "id" from file "file"
        String id = cr.readCardFile("id", "file");

        System.out.println(id);
    }

    // handle exceptions
    catch (CardException e)
    {
        System.err.println(e.getMessage());
    }
}

107
SmartCard sc =
    SmartCard.waitForCard(cr);

FileAccessCardService facs =
    (FileAccessCardService)
    sc.getCardService(
        FileAccessCardService.class, true);
CardFile root = new CardFile(facs);
CardFile file =
    new CardFile(root, ":6964");

byte[] data =
    facs.read(file.getPath(), 0,
            file.getLength() );
sc.close();

String entry = new String(data);
entry = entry.trim();
System.out.println(entry);

} catch (Exception e) {
    e.printStackTrace(System.err);
}

} finally {
    // even in case of an error
    try {
        SmartCard.shutdown();

108
The example code is easy to understand for those familiar with Java. Programmers can take advantage of the higher abstraction of Java, such as I/O streams, etc. OCF is integrated with JavaCards, providing a consistent development environment for application software and on-chip software.

However, the reliance on Java is also a disadvantage. First, Java may be viewed as overkill for many smartcard applications, as smartcards are small and simple devices. In SCFS, I use a smartcard in a simple way. For example, I can print out a file (as in the OCF example) by typing:

```
% mount_scfs /dev/scfs0 /smartcard
% cat /smartcard/id
```

Second, being written in Java, OCF is typically much slower than C based middleware, such as the sc7816 library [85].

**PC/SC**

PC/SC is a general purpose architecture for integrating a smartcard into PCs [22]. It provides abstraction in a level lower than SCFS. The strong advantage of PC/SC is that its smartcard reader driver interface, IFD handler [27], is supported widely.
To take advantage of this, the IFD handler interface has been integrated into sc7816. This allows SCFS to take advantage of many smartcard reader drivers written as an IFD handler.

Special Purpose Standards

Application specific standards such as PKCS#11, EMV, and SET have advantages in usability in specific domains because of higher abstractions than SCFS. In SCFS, functionality to take advantage of smartcard security, such as internal and external authentication, is given by the ioctl() system call. However, ioctl() is not as user friendly as the functionality provided by PKCS#11, EMV, and so on. I may provide libraries for specific purposes to wrap around SCFS to give higher abstractions.

4.1.6 Section Summary

I have implemented a Smartcard Filesystem (SCFS) to ease development of smartcard software. SCFS provides a UNIX filesystem API for a smartcard. Developers can use the well-established UNIX API and development environment for smartcards. Performance evaluation shows the overhead caused by SCFS is negligible.

The following three aspects highlight the value of this work.

Transparent API with the UNIX Filesystem

SCFS differs from the other approaches such as OCF and PC/SC in that it is implemented as an operating system extension. To an application, smartcard files look identical to files stored on other media. With SCFS, an application can use a smartcard without modification (Figure 4.5).

With SCFS, many UNIX applications can take advantage of smartcard security without modification. For example, here is how I made SSH work with a private
key stored in a smartcard: I added a symbolic link from $HOME/.ssh/identity to
/ smartcard/ss/id and copied a private-key to the SSH identity file.

citi% mount_scfs /dev/scfs0 /smartcard

citi% ln -s /smartcard/ss/id ~/.ssh/identity

citi% ssh sin.citi.umich.edu

Enter PIN:

sin% logout

PGP works with a private key in a smartcard in a similar way:

citi% mv ~/.pgp/secring.pgp /smartcard/pg/ky

citi% ln -s /smartcard/pg/ky ~/.pgp/secring.pgp

Kerberos tickets and browser cookies can be stored in SCFS in similar ways.

In contrast, OCF or PC/SC require that an application be modified to use a
smartcard because the API for a smartcard is different from the API for normal files
(Figure 4.6).

Figure 4.5: Application that uses SCFS

Figure 4.6: Application that uses OCF or PC/SC
Portability

Another advantage of SCFS is portability. Most of the SCFS code is in user space and easily ported to other operating systems. The xfs kernel module is based on Arla, which is already ported to many UNIX-like operating systems, including Solaris, NetBSD, FreeBSD, OpenBSD, Linux, AIX, HP-UX and Digital UNIX. It is easy to port SCFS xfs to other operating systems.

Useful Development Tool

Smartcard standards other than SCFS give higher abstractions for users, e.g., Java language in OCF, EMV’96 for electric commerce, PKCS#11 for cryptographic applications. Depending on the type of applications, different kinds of abstraction may be required. Therefore, there are many standards that do not interoperate [11]. In contrast, SCFS works with a raw smartcard with a minimum amount of abstraction; no matter what functionality a smartcard offers, SCFS can access and use its secure storage. SCFS is especially helpful in maintenance, testing, and debugging. For example, SCFS is used to maintain the key table in the Kerberos / smartcard integration project (Section 3.1) and homepage contents in the webcard project [86].
4.2 Smartcard Secured Cryptographic File System

4.2.1 Introduction

An obvious countermeasure to theft of secrets stored in computers is to encrypt the secrets, and store the encryption key somewhere else. Matt Blaze has realized this with his Cryptographic File System for UNIX (CFS)\(^6\), which transparently encrypts files in a file system [9]. Although CFS adds significant security to current systems, its weakness arises in key management: it relies on a user chosen password. I point out two problems with using passwords in CFS. First, as I repeatedly mention, user chosen passwords are often vulnerable to dictionary attack. Once an adversary obtains the ciphertext by stealing a computer or by compromising it, he can run off-line dictionary attack. Second, because a user cannot remember many passwords, the number of passwords is limited. CFS employs a key per directory-tree policy, which is not as desirable as a key per file policy. If a key is stolen, e.g., by compromising an administrator account or by obtaining it from virtual memory backing store [83], the files encrypted under the key are revealed. Therefore, the fewer files are encrypted under a single key, the better.

I address these problems by storing a randomly generated user key on a smartcard, and by using it to generate a file key that is used to encrypt only one file. This prevents dictionary attack, and minimizes the number of the file stolen on host compromise. I have implemented such a system called SC-CFS, based on CFS. The design, security consideration, implementation, and performance evaluation are discussed in this section.

\(^6\)Throughout this document, I refer to CFS version 1.3.3 by “CFS”.

113
4.2.2 Design

Cryptographic File System Review

Because 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 [72] server daemon (namely, it provides a file system that can be mounted and be accessed through the NFS protocol Version 2) 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 [9].

Design Goals

The goals of SC-CFS 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 access the files in the file system. Therefore, a file key, which is used to encrypt and decrypt a file, should be derivable from the master key in the smartcard. On the other hand, the master key must NOT be derived 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 4.2.3.

- The file key changes when a file changes.

  When a file is written, its associated file key changes to protect the new file content. Consider the following scenario: The 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.

**Design**

I designed the following key management scheme to achieve the goals.

- A randomly generated master key is stored in a smartcard.
- **cfsd** stores a file’s inode number and a time-stamp 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. It replies with the SHA1 hash result of the seed concatenated with the master key. (SHA1\{inode#, timestamp, K_{user}\}). This is 20 byte long.
- **cfsd** further hashes the result into 8 byte DES key, and this is the file key to encrypt and decrypt the file.

**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 directory. When a user starts accessing the directory with **cattach**, it decrypts the signature. If it recovers the predefined string correctly, the user entered the right password (in CFS) or used the right smartcard (in SC-CFS), and he is authenticated to enter the directory.
In SC-CFS, before the smartcard is used, a 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 number of wrong PINs are entered.

**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 7. 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.

I cannot use this partial encryption approach, as one of my goals is to change a key every time a file changes. Therefore, an entire file is re-encrypted on modification. 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. I cannot afford to change a file key and encrypt the entire file 256 times.

To solve this problem, 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 (the NFS server is stateless), write back happens in one of the

---

7A *directory key* is a key used to encrypt files in a directory. This is entered by a user.
following events:

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

4.2.3 Security Consideration

I discuss the security of my approach here, mainly in comparison with CFS. Another cryptographic file system, Transparent Cryptographic File System (or TCFS) [14], has a key management system similar to CFS. Discussion about CFS in this section also applies to TCFS.

Model

I start with constructing a model of the system. It 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.
Threats

I make the following assumptions in the model.

1. Mallory can compromise the host.
   Mallory can read and modify any information on the host.

2. Mallory can change behavior of the host, but it is difficult.
   By Assumption 1, Mallory is able to install a Trojan horse in the host, which, for example, steals file keys. However, I assume this attack is difficult.

3. Mallory cannot compromise the smartcard.
   Mallory can neither read nor modify any information on the smartcard. She cannot influence the behavior of the smartcard, either.

4. Cryptographic operations are strong.
   My principal cipher is DES, which is assumed impossible to compromise in reasonable amount of time. Also, my principal hash function, SHA1, is assumed to be collision free.

Attack

Directory / File Key Theft

If the host is compromised (possible by Assumption 1), the keys currently used (thus in memory) can be stolen in both 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. Until the key is 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 user master key is safe because it
is in a smartcard (Assumption 3). When updated, the file will be encrypted under a different key, so the file is safe after it is updated.

SC-CFS is more secure than CFS because when a key is stolen, only one file can be decrypted by the key. In fact, 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 under the directory, 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 the randomly generated master key, and generates file keys, eliminating this problem.

**Master Key Theft**

In CFS, when a host is compromised, the master key may be recovered, as it may remain in the process memory, or it may remain in the virtual memory backing store. Niels Provos has pointed out that virtual memory backing store may contain critical secrets even though application programs delete them [83]. By reading a hard disk which is used as the backing store, Mallory may be able to recover the secrets.

In SC-CFS, the master key is in a smartcard, and cannot be stolen.

**Storage Theft**

Storage theft is sometimes more easily executed 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 in CFS.

• Apply reversed CFS encryption operation to an encrypted file.

• If this recovers a readable text, this is almost certainly the right key. If it does not, pick another password and try again.

Many sophisticated password crackers are published (e.g., John the Ripper [82]), and should be useful to implement such an attack.

In SC-CFS, the master key is a random number, and is not vulnerable to dictionary attack. By Assumption 4, brute force attack for the master key is impossible, too.

**On-Line Attack**

In both CFS and SC-CFS, user authentication is done at *cattach*, with a password in CFS and with a PIN in SC-CFS. As a consequence, if Mallory compromises a host (possible by Assumption 1) while Alice is using CFS or SC-CFS, or installs a Trojan horse that waits until Alice logs in (possible by Assumption 2), Mallory 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, if a LED box that indicates data transmission via a serial port is installed on Alice’s computer, she knows it when Mallory is accessing her files illegally.

**4.2.4 Implementation**

Host-side implementation is tested on Linux-2.2 and OpenBSD-2.6. NFS Version 2 is a standard protocol that runs on almost any UNIX. Smartcard-side implementation is specific to Schlumberger Cyberflex Access smartcard. Because Cyberflex Access is a Java card, I refer to the smartcard-side program as “SC-CFS applet”.

120
SC-CFS has been implemented as an extension to CFS. The implementation is divided into the following parts: modification to \texttt{cfssd}, \texttt{cattach}, \texttt{cmkdir}, and implementation of the SC-CFS applet. Here I discuss each part.

- **Modification to \texttt{cfssd}**

  In CFS, \texttt{cfssd} stores \{inode\#, creation time\} in a file called \texttt{.pvec\_encrypted\_filename}.\footnote{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 \texttt{touch}.} First, \texttt{cfssd} 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. Second, the single-file cache described in Section 4.2.2 is implemented. Finally, \texttt{read} and \texttt{write} operations are modified to access data through the cache.

- **Modification to \texttt{cattach}**

  When \texttt{cattach} is invoked with \texttt{-p port\_number} option, it asks for a PIN instead of a password and then sends it to \texttt{cfssd}. \texttt{cfssd} initializes the smartcard, sends the PIN to the smartcard, and then carries out the card authentication described in Section 4.2.2.

- **Modification to \texttt{cdetach}**

  When \texttt{cdetach} is invoked, \texttt{cfssd} cleans up the cache and terminates the connection with the smartcard.

- **Modification to \texttt{cmkdir}**

  When \texttt{cmkdir} is invoked with \texttt{-S} option, it creates a signature described in Section 4.2.2 in the newly created SC-CFS directory, using the key generated by the smartcard.
• 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.

4.2.5 Performance Evaluation

I have evaluated the performance of SC-CFS in comparison with CFS and a local file system (Linux EXT2 File System). First, the result of the Andrew Benchmark Test [48] is reported to show the user response time. Then, I 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 the smartcard to generates keys. Most of this penalty is due to the slowness 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. Variance is small.

Round Trip Time

The Andrew Benchmark, a standard file system benchmark test, is used to measure the round trip time of SC-CFS. The benchmark has five phases: MakeDir (mkdir), Copy (cp), ScanDir (ls -1), 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 \(^9\). The result is shown in Table 4.4.

\(^9\)I added five global variables, removed two getchar()s, and changed options to ar. None of them should alter performance significantly.
<table>
<thead>
<tr>
<th></th>
<th>Local (s)</th>
<th>CFS (s)</th>
<th>SC-CFS (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MakeDir</td>
<td>0</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>Copy</td>
<td>0.6</td>
<td>1.0</td>
<td>21.8</td>
</tr>
<tr>
<td>ScanDir</td>
<td>1.2</td>
<td>1.6</td>
<td>1.0</td>
</tr>
<tr>
<td>ReadAll</td>
<td>2.0</td>
<td>3.0</td>
<td>22.6</td>
</tr>
<tr>
<td>Make</td>
<td>5.0</td>
<td>7.8</td>
<td>29.6</td>
</tr>
</tbody>
</table>

Table 4.4: SC-CFS Andrew Benchmark Result

SC-CFS works as efficiently as Local and CFS when it does not need to access a smartcard (MakeDir and ScanDir\(^{10}\)). 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. Table 4.5 shows: (1) the number of accesses to a smartcard, (2) (1) × 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 can be attributed to the smartcard.

<table>
<thead>
<tr>
<th></th>
<th>#access</th>
<th>× 0.31</th>
<th>SC-CFS - CFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MakeDir</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Copy</td>
<td>70</td>
<td>21.7</td>
<td>20.8</td>
</tr>
<tr>
<td>ScanDir</td>
<td>0</td>
<td>0</td>
<td>-0.6</td>
</tr>
<tr>
<td>ReadAll</td>
<td>70</td>
<td>21.7</td>
<td>19.6</td>
</tr>
<tr>
<td>Make</td>
<td>75</td>
<td>23.3</td>
<td>21.8</td>
</tr>
</tbody>
</table>

Table 4.5: SC-CFS Smartcard Performance Impact

**Detailed Look**

As smartcard access is shown to be the bottleneck of SC-CFS, this part deserves special attention. Detailed performance evaluation was carried out on Cyberflex Access, which communicates at 38.4 Kbps with the host.

\(^{10}\)File attributes retrieved by `stat()` are not encrypted.
SC-CFS’s smartcard operation involves two APDUs: 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 [50] defines the T=0 communication protocol, which Cyberflex Access adopts, to be uni-directional, i.e., a smartcard can either send or receive data on one APDU. Therefore, in addition to `generate_key` APDU, `get_response` APDU is necessary. These two APDUs are sent to the smartcard consecutively.

Table 4.6 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.

<table>
<thead>
<tr>
<th>operation</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash (SHA1) 24 byte into 20 byte</td>
<td>0.15</td>
</tr>
<tr>
<td><code>generate_key</code> APDU overhead</td>
<td>0.10</td>
</tr>
<tr>
<td>Select root in file system</td>
<td>0.01</td>
</tr>
<tr>
<td>Select key file “ke” in file system</td>
<td>0.01</td>
</tr>
<tr>
<td>Read 16 byte from key file</td>
<td>0.01</td>
</tr>
<tr>
<td><code>get_response</code> APDU (20 byte)</td>
<td>0.01</td>
</tr>
<tr>
<td>total</td>
<td>0.29</td>
</tr>
</tbody>
</table>

Table 4.6: SC-CFS Smartcard Performance Breakdown

The cost boils down to two dominating operations: SHA1 hash function and `generate_key` APDU overhead. These two are necessary operations, and I 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.
4.2.6 Discussion

Administration Tools

With the current SC-CFS prototype, a user has to manually update his master key and PIN via CITI’s smartcard communication tool called pay [85]. Automated tools to do this should be provided.

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 4.2.5, the overhead is dominated by individual operations in a smartcard, which I, as an application developer, cannot change. I hope new smartcards or other similar devices will achieve much higher performance in the near future.

Kerberized Cryptographic File System

I observe that Kerberos [59] can provide a similar functionality with a smartcard in CFS, using a Kerberos service for key generation. The following is a rough design of this system (K-CFS):

- Create a new Kerberos service, the K-CFS key server.

- The key server stores a user’s master key ($K_{user}$).

- The user establishes an encrypted channel with the key server through Kerberos authentication service and ticket granting service.

125
• The user requests a file key to the key server by sending \{inode\#, timestamp\} to the server.

• The server returns the file key: \{inode\#, timestamp\}K_{user}.

• K-CFS uses the file key for file encryption.

K-CFS has similar security properties with SC-CFS:

• Each file is encrypted under a unique key.

• Host compromise will yield only opened files.

• Storage theft will not reveal files, as the user master key is not vulnerable to dictionary attack.

The difference between SC-CFS and K-CFS is the trust models. In SC-CFS, a user trusts her personal belonging, a smartcard. In K-CFS, she trusts the Kerberos server administrators and the Kerberos client software.

An advantage of K-CFS over SC-CFS is that a smartcard access infrastructure is not necessary. The disadvantage is that it is vulnerable if the Kerberos infrastructure is compromised. By compromising a client host or a server host, an adversary can obtain a ticket and a session key, making impersonation possible. In addition, K-CFS requires network connection, making it hard to use on laptops.

Depending on infrastructure and assumptions users have, K-CFS may be useful as well.

4.2.7 Section Summary

I 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 1.2.1, the increasing threat of computer theft demands a way to protect secrets on a computer. CFS is a secure and seamless solution to this problem, and 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.

As in authentication (Chapter 3) and SCFS (Section 4.1), this is a new, interesting application for smartcards. I believe that widespread deployment of smartcards is essential to the security of computer systems: applications such as this one could be a driving force for smartcard deployment.

• Remark on smartcard performance

Performance evaluation in Section 4.2.5 once again shows the importance of native smartcard performance. In recent years, smartcards have matured in terms of functionality and reliability. However, I have not seen significant performance improvement, even though microprocessors have sped up by 5 to 10 times.
Chapter 5

Personal Secure Booting with Smartcards

This chapter again attacks the problem of trusted computing. In the previous two chapters, I explored ways to make security systems survive even where underlying computers are not trustworthy. In this chapter, I take a different approach, that is, I improve an existing secure booting system for personalization and configurability.

Secure booting for a workstation was developed in the AEGIS project by William Arbaugh et al. [3, 2]. AEGIS already has secure hardware in a form of a read-only memory chip which stores the first program executed in the bootstrap process, and the public key used through the process. Because this chip is not writable by an adversary, AEGIS is secure against modification attempt to other portion of the workstation.

The problem with AEGIS is that a user must trust his system administrator to provide trusted operating systems and applications. However, because (1) security threats often come from inside of organizations, and (2) in public computing sites, such as Internet cafes, system administrators are unknown, the user may choose to not trust system administrators. I embrace a smartcard as personal storage of signatures to hand the control of the system to the user.
The smartcard integrated system, called *sAEGIS*, is reported in this chapter. Computer components, e.g., operating systems and applications, are not modified for the ease of deployment.

5.1 Personal Secure Booting

5.1.1 Introduction

This section reports *sAEGIS*, a smartcard integration to AEGIS secure bootstrap process. This adds personalization, authentication, and flexibility to AEGIS. It takes advantage of *GRUB*, a free, powerful boot loader [31] which can boot various kinds of operating systems.

The design, security properties, implementation, and performance evaluation of sAEGIS are presented in this section.

5.1.2 Design

Design Goals

The goal of sAEGIS is as follows:

- Personalization

  In AEGIS, it is a system administrator’s responsibility to manage MACs. However, because (1) security threats often come from inside of organizations, and (2) in public computing sites, such as Internet cafes, system administrators are unknown, users may choose to not trust system administrators. By embracing a smartcard as personal storage of MACs, sAEGIS hands the control to the users.

- Authentication
In AEGIS, a user who attempts to boot a computer is not authenticated. That is, anyone who can invoke the boot process, for example, by hitting the reset button, may boot it. sAEGIS boots an operating system only if a correct smartcard and associated PIN are presented by a user. This two-level authentication (what-you-have and what-you-know) makes theft of a mobile computer less threatening, as the thief cannot use the computer.

- Operating System Flexibility

The only operating system the AEGIS prototype is able to boot is FreeBSD. sAEGIS, in contrast, adopts a free, flexible boot loader called GRUB [31] to boot several operating systems, namely, Linux, FreeBSD, NetBSD, OpenBSD, Windows 9*, NT, and 2000.

- Hardware Configuration Flexibility

Because of the size limitation of BIOS, it is hard to fit file system drivers into BIOS. As a result, a program in BIOS cannot access data in a hard disk. Therefore, MACs of extension cards, such as a video card and a network card, must be in BIOS, instead of in a hard disk. This hampers hardware configuration, as every configuration change (replacement of an extension card, for example) causes modification of a BIOS image, forcing reprogramming of the flash chip that contains the BIOS. Because the smartcard access library is small enough to fit into the BIOS, the hardware configuration information and MACs can be moved to the smartcard, removing the necessity of reprogramming the flash chip.

In the above four goals, the first three were achieved in my prototype. The reason why the last goal was not achieved is discussed in Section 5.1.6.
Design Overview

Overview of the design of sAEGIS is given in this section.

In a nutshell, sAEGIS = AEGIS + GRUB + smartcard + verify. That is, (1) sAEGIS relies on AEGIS to boot GRUB securely, (2) GRUB boots an operating system kernel securely using a smartcard for verification, and (3) the kernel checks the integrity of daemons with an application called verify. I do not present details of AEGIS internals in this dissertation. Interested readers are advised to consult with articles [2, 3].

The basic idea behind sAEGIS is as follows: if a lower layer verifies the integrity of all higher layers before booting them, the system integrity is assured. So, to comprehend the design of sAEGIS, it is essential to understand which component verifies and boots whom, and how. The bootstrap process of sAEGIS is summarized in the following events, in chronological order.

1. Power on Self Test (POST). The processor checks itself.

   POST is invoked by either applying power to the computer, hardware reset, warm boot (ctrl-alt-del under DOS), or jump to the processor reset vector invoked by software. This starts the boot strap process.

2. BIOS section 1 verifies the MAC of BIOS section 2, and boots it.

   In sAEGIS, BIOS is divided into two parts, section 1 and section 2. The former contains the bare essentials needed for integrity verification, such as a cryptographic hash function (MD5 and SHA1), a public key function (RSA), and the MAC of the BIOS stage 2.

   The integrity of this part is assumed, i.e., it is assumed to never be modified. More discussion about this assumption is in Section 5.1.3.

131
3. BIOS section 2 verifies the MACs of ROM of extension cards, and executes them.

Before passing control to the programs stored in the ROM of the extension cards, BIOS section 2 verifies the integrity of these programs. The MACs are stored in the BIOS section 2.

4. BIOS section 2 verifies the MAC of GRUB stage 1, and boots it.

GRUB is divided into two parts, stage 1 and stage 2, because an Intel-compatible personal computer requires a primary boot loader to be no more than 512 byte long. Stage 1 is booted by BIOS section 2; and stage 2 is booted by stage 1. In sAEGIS prototype, both stage 1 and stage 2 of GRUB are stored in a floppy disk. The MAC of GRUB stage 1 is stored in the BIOS section 2.

5. GRUB stage 1 verifies the MAC of GRUB stage 2, and boots it.

Before booting stage 2, stage 1 verifies it. Both the program and its MAC are in a floppy disk.

6. GRUB stage 2 verifies the MACs of the kernel and the verification tools, and boots the kernel.

GRUB stage 2 mounts a file system (typically on a hard disk) where a kernel, verify application, and a shell script which invoke verify (e.g., /etc/rc.d/init.d/inet on UNIX) are on. It reads these files and verifies the MACs of them before booting the kernel. The MACs are stored in the smartcard.

7. The kernel uses verify application to verify the important files, and starts system daemons that pass the check.
verify is invoked by the kernel at boot to check important files. If the check fails, the kernel does not start the related daemons. The important files are system daemons (e.g., login, logind, ssh, and sshd should be verified on UNIX to detect a password sniffer), configuration files (e.g., SYSTEM.INI should be verified on Windows to detect a Trojan horse), and shared libraries (e.g., GINA.DLL should be verified on Windows NT / 2000 to detect a password sniffer).

The boot strap process is depicted in Figure 5.1.

![Diagram](image)

Figure 5.1: Boot Strap Process

**Smartcard Communication Protocol**

In the step 6 of the list presented above, a workstation and a smartcard carry out a protocol to (1) authenticate the smartcard and (2) verify the hash presented by the workstation. The protocol is shown in Figure 5.2, and is described as follows.
Workstation:

- obtain PIN from user
- compute hash of kernel: \( m = \text{SHA1\{kernel\}} \)
- generate random challenge: \( r \)
- encrypt \( \{m, r\} \) with public key: \( \{m, r\}K_{\text{pub}} \)
- send \( \{m, r\}K_{\text{pub}} \) to smartcard, along with the PIN

Smartcard:

- check PIN. If the PIN does not match, set ANSWER to ERR.
- decrypt \( \{m, r\} \) with private key
- compare \( m \) to its stored hash, and set ANSWER to OK or ERR
- sign \( \{\text{ANSWER}, r, m\} \) with Kprv: \( \{\text{ANSWER}, r, m\}K_{\text{prv}} \)
- send it to workstation

Workstation:

- encrypt \( \{\text{ANSWER}, r\}K_{\text{prv}} \) with \( K_{\text{pub}} \)
- make sure it is signed by a smartcard.

- if (ANSWER == OK and \( r \) == original \( r \) and \( m \) == original \( m \)) continue with boot. Otherwise, halt the boot process.

### 5.1.3 Security Consideration

In this section, I discuss the security of the design presented in Section 5.1.2.
Figure 5.2: Smartcard - Workstation Communication Protocol

Model

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

**Alice (A)** A legitimate user who wants to boot and use a PC. She owns a smartcard.

**Smartcard** Alice’s smartcard. It stores a private key, $K_{prv}$, and MACs. It is PIN protected, i.e., a secret number must be presented before it is used, and it blocks itself if a wrong PIN is typed for $n$ consecutive times.

**Mallory (M)** An adversary.

**Personal Computer (PC)** An Intel-compatible personal computer to be verified and be booted. It consists with BIOS section 1 and 2, extension cards, GRUB boot loader stage 1 and 2, an operating system kernel, **verify**, and the other files.

Claims

Here I claim the security properties of my system.

System integrity after boot
When a PC is booted using sAEGIS, the integrity of the following components of the PC are ensured; BIOS, extension cards, GRUB boot loader, operating system, and other files that are verified.

User authentication

When a PC is booted using sAEGIS, it has been booted by a legitimate user.

Threats

I make the following assumptions in my model.

1. BIOS section 1 is integral.

   The security property of the entire sAEGIS system relies on this assumption, as BIOS section 1 is the base of the secure boot strap. If BIOS section 1 is modified maliciously, BIOS section 2 may not be verified correctly, resulting an unintegral section 2. This leads to an unintegral GRUB stage 1, stage 2, and finally, an unintegral operating system kernel. So this defeats the goal of sAEGIS.

   I believe this assumption is reasonable. A portion of Intel’s latest generation of flash ROM can be write protected by setting one of the PINs (RP#) to high [44]. Furthermore, if the chip set vendors choose to do so, BIOS section 1 can be stored in ROM, prohibiting any modification.

2. Mallory can read anything in PC, but nothing in smartcard.

   Mallory can read any data stored in the PC. However, she cannot read any data in the smartcard. This is a reasonable assumption, as it is usually easy to physically open a PC and access data storage in it. In contrast, a smartcard is a tamper-resistant device.
3. Mallory can write anything in PC except in BIOS section 1. She cannot write anything in smartcard.

Similarly to Assumption 2, Mallory can write anything in PC except in the protected region. However, she cannot write anything in the smartcard.

4. Cryptographic functions are strong.

I assume that cryptographic hash functions (MD5 used in BIOS, and SHA1 used in GRUB stage 2) are collision-free. I also assume that the random number generator used in the protocol given in Section 5.1.2 is unpredictable. Finally, I assume that my principal cipher, RSA, is impossible to compromise in reasonable amount of time.

5. Mallory does not know Alice’s private key.

6. Mallory can snoop and modify messages on the serial port in which the PC and the smartcard are communicating.

**Attacks**

**Modification to PC’s components**

By Assumption 3, Mallory can modify anything she wants in the host except the BIOS section 1. However, if she does so, Alice will notice it at the next boot, as sAEGIS verifies every byte of code that are executed during the boot strap process.

By Assumption 1, a correct boot strap process will be invoked every time Alice boots the PC. By Assumption 4, Mallory cannot forge a MAC without knowing Alice’s private key, and this does not happen by Assumption 5.
Modification to PC’s components after boot

Being a secure boot strap system, sAEGIS makes no attempt to protect the PC after it is booted. Mallory can modify the system maliciously, e.g., install Trojan horse or a sniffer. However, Alice can always bring the PC back to the integral state by rebooting it.

Unauthorized Boot Attempt

Mallory may steal the PC and try to use it. This is impossible unless Mallory obtains Alice’s smartcard and PIN, as the authentication protocol presented in Section 5.1.2 prevents such an attempt. Without knowing Alice’s private key, \( K_{prv} \) (Assumption 2 and 5), Mallory cannot produce \( \{ OK, r\} K_{prv} \), because the random number generator is strong (Assumption 4).

Mallory may try to reply an OK message \( \{ OK, r\} K_{prv} \), but this does not work, either, because of the random nonce, \( r \).

Mallory may try man-in-the-middle attack, i.e., modifying the kernel and replacing the message from the host, \( \{ m', r\} K_{pub} \), with \( \{ m, r\} K_{pub} \). The smartcard, not knowing the hash value was altered, sends OK message. However, the workstation notices the attack because the hash values \( m' \) and \( m \) do not match.

Serial Cable Wiretapping

By Assumption 6, Mallory can read and write messages on the serial cable connecting the PC and the smartcard. However, she cannot produce \( \{ OK, r\} K_{prv} \).

Mallory as System Administrator

Mallory may be Alice’s malicious system administrator, and may try to compromise her secrets. For example, consider a case in which Mallory tries to read Alice’s e-mail.
Alice may encrypt her e-mail with a secure mail tool, e.g., PGP. Without a system like sAEGIS, Mallory can modify the executable code of PGP to leak information for her. sAEGIS prevents this by detecting such modification. If operating system and application software vendors publish the signatures of their software, Alice can store the signatures in her smartcard, and can check the system.

It is still unclear whether I can counter all the possible attacks mounted by system administrators, as security software usually is written in the assumption that system administrators are trustworthy, and attacks by system administrators have not been well studied. However, I believe that sAEGIS is the first step to counter such attacks.

**Chosen Ciphertext Attack**

Chosen ciphertext attacks on RSA exist [89]. In sAEGIS, however, the value signed by the smartcard is different from the value sent to the smartcard, making the attack impossible.

### 5.1.4 Implementation

I describe the sAEGIS prototype, which is an implementation of the design described in Section 5.1.2. It is implemented on an ASUS P55T2P4 Pentium motherboard, running 233 MHz AMD K6 processor.

The prototype is based on the AEGIS prototype by Arbaugh et al. I do not go into the details of AEGIS implementation. sAEGIS is also based on GNU GRUB 0.5.93.1. Again, I do not explain the details of GRUB. Interested readers are advised to visit [31].

**GRUB stage 1**

GRUB stage 1 is modified to verify GRUB stage 2 before jumping to it. stage 1 tells AEGIS where stage 2 starts (0x800:0) and how large it is, and calls the AEGIS
interrupt (0xc2).

**GRUB stage 2**

GRUB stage 2 is modified to carry out the protocol described in Section 5.1.2.

First, to communicate with a smartcard through a serial port, the smartcard communication library is implemented by replacing the system dependent part of sc7816 library [85] with modified serial console access routines in OpenBSD-2.4 (/usr/src/sys/dev/ic/com.c).

Then, it needs some cryptographic functions. SHA1 routines in GRUB are ported from Kerberos version 5-1.0.5 distributed by MIT. RSA routines are taken from PGP 2.6.2.

In this prototype, random number generation is not implemented. It is replaced with a constant.

kernel command in the GRUB user interface loads a kernel from a file system to main memory. This command is modified to invoke the verification protocol before letting GRUB boot the kernel. Another command, “updatehash”, is added to update SHA1 hash so that files can be verified in addition to the kernel.

**verify**

**verify** is a C program that reads a given file, computes its hash, and verifies it with a hash stored in a file, and returns the result of verification. An example use of **verify** is as follows. In this example, **verify** makes sure **inetc** is not modified before it is started.

/etc/rc.d/init.d/inet:

    /boot/verify /usr/sbin/inetc /boot/hash-table.txt &

daemon /usr/sbin/inetd

In the future implementation, verify should use hashes stored in a smartcard.

**Smartcard Side Code**

The program in a smartcard is implemented in a Schlumberger Cyberflex Access smartcard with Java. Cyberflex Access is the only smartcard I know that offers both programmability and cryptographic functions (DES, RSA, and SHA1).

The smartcard reads 128 byte input from GRUB, decrypts it with the RSA private key. It then compares the hash value with the one that has been previously stored in its memory and determines whether the kernel image is integral. It concatenates its reply (0x8080808080808080 if OK, 0x4040404040404040 if not) with the random key and signs the resulting string with the RSA private key. Finally, it sends the result to GRUB.

In this prototype, a smartcard can hold only one SHA1 hash value. This should be improved to allow more flexibility.

**5.1.5 Performance Evaluation**

To evaluate the efficiency of sAEGIS, the boot process is timed. The following is the amount of time elapsed since a PC is powered up until an operating system starts the last system daemon. In addition, the smartcard access time (the time spent in the protocol in Section 5.1.2) is measured, as it is one of the most expensive components.

Measurement was carried out on Linux 2.2 (RedHat 6.2) with 233 MHz AMD K6 processor. I used the RDTSC instruction to obtain the number of ticks after the processor powers up. All the numbers are in seconds, and are average of 5 trials. Variance is small.
<table>
<thead>
<tr>
<th></th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>boot with sAEGIS</td>
<td>69.55</td>
</tr>
<tr>
<td>boot without sAEGIS</td>
<td>57.88</td>
</tr>
<tr>
<td>difference</td>
<td>11.67</td>
</tr>
<tr>
<td>smartcard access</td>
<td>5.54</td>
</tr>
</tbody>
</table>

The result shows that sAEGIS adds 11.67 seconds. About half of the added cost is for accessing the smartcard. The other half includes the following:

- Code checking, which involves MD5 hashing and RSA operations. More details in this are available in [2].

- Loading GRUB, which is 77KB large, from a floppy disk, takes more time than loading the much smaller (4.5KB) Linux boot loader, LILO from a hard disk.

Adding 11.67 seconds to the bootstrap process, which already takes 1 minute, should be acceptable in many environments.

5.1.6 Discussion

Key Management

To use sAEGIS effectively, it is essential to manage the private key in the smartcard correctly. For the computers that are personal, e.g., laptop computers, the private key should be known only to the owner of the computer. This discourages an adversary from stealing computers because he cannot use them without also stealing the smartcards. For the computers that are public, e.g., workstations in a library, the private key should be shared between the users. Each user can set up MACs in her smartcard using the private key.
Fix Implementation Limitations

Three implementation limitations described in Section 5.1.4 should be fixed, namely, (1) no random number generator, (2) verify does not use a hash in the smartcard, (3) kernel hash, $m$, is not included in the message the smartcard sends to the workstation, and (4) the smartcard holds only one hash.

Smartcard in BIOS

To achieve Goal 4 described in Section 5.1.1, it is necessary to move the smartcard access library into BIOS. The library is 11 KB, so the size should not be a problem for the 1M flash BIOS.

Unfortunately, I did not have permission to access the BIOS source code. Instead of working out licensing issues, I decided to publish a prototype, and to wait until open source BIOS projects are mature enough to be used as the next platform [66, 79].

5.1.7 Section Summary

I have implemented a personal, secure boot strap process, sAEGIS, which is an extension to AEGIS. Advantages of sAEGIS over AEGIS are: (1) the smartcard lets a user control what he uses, (2) the smartcard serves as an authentication token, and (3) it is more flexible than AEGIS.

The following two aspects highlight the value of this work.

- Improvement to important software.

Secure bootstrap, like AEGIS, is strongly demanded, as attacks that modify the operating system are becoming more common [40]. One of the problems of AEGIS is the lack of flexibility: it can boot only the FreeBSD kernel, and it requires reprogramming of EEPROM chip when hardware configuration is changed. I solved the former problem, and proposed a solution to the latter.
• Idea of personalization.

sAEGIS suggests a system in which the user does not have to trust his system administrators. I think it is a huge security gain, as many attacks come from inside organizations.
Chapter 6

Conclusion

6.1 Summary

The security of modern computer systems can be greatly enhanced by integrating secure hardware into current security systems. This dissertation showed this by developing and evaluating such systems.

I first pointed out that:

- Modern computer systems cannot be trusted because of the lack of physical security and software bug exploitation.

- Reliance of passwords can be exploited.

I first argued the significance of these problems. Then I proposed a solution: store secrets and use them in secure hardware. This way, the secrets are safe even under hostile condition. I then applied this solution to important security applications: authentication, secure storage, and secure booting.

In Chapter 3, I presented secure hardware integration with Kerberos. The summary of this project is as follows:

- The smartcard integrated Kerberos client is more secure than the original, as (1) it does not give up a user’s master key and a TGT session key on host
compromise, and (2) it prevents dictionary attack.

- The secure coprocessor integrated Kerberos server is more secure than the original, as (1) it does not give up a realm’s master key, principals’ master keys, or session keys on host compromise, and (2) it prevents ticket forgery.

- The smartcard integrated Kerberos is enhanced so that a user can use his smartcard even if it is attached to a remote workstation. This improves the usability of smartcards, and yet does not weaken the security features.

In Chapter 4, I developed two secure hardware based secure storage systems:

- Smartcard Filesystem provides a convenient interface for smartcards through UNIX file system abstraction, providing easily accessible secure storage.

- Matt Blaze’s Cryptographic File System is improved by smartcard integration. The new file system is more secure than CFS because: (1) it does not give up the user’s master key on host compromise; (2) it does not give up encryption keys for unopened files on host compromise; and (3) it prevents dictionary attack.

Finally, Chapter 5 presented secure hardware based secure booting system:

- A smartcard is integrated into William Arbaugh’s AEGIS secure bootstrap process. In addition to AEGIS’s protection against external adversaries, the new system prevents internal adversaries, e.g., system administrators, from modifying computer systems maliciously.

### 6.2 Contribution

The following highlights the contribution of this work:
• Improvement to important security systems

Kerberos, CFS, and AEGIS are all very important security systems. Kerberos’ user base is large and will continue to expand, as Microsoft embraced it as the default network authentication mechanism. Although not as widely used as Kerberos, CFS and AEGIS’ contributions to secure storage and secure booting will have an important role in the future of computing. Secure hardware integration improves the security of all these systems.

• Killer applications for secure hardware

It is widely accepted that secure hardware is a good thing, but it has not yet been widely deployed. This is because the initial investment is high, and there are not enough applications to convince people to start using them. I believe my policy, using secure hardware in information technology systems, is a step in the right direction, and I hope this dissertation will spark many more ideas and applications.

• Detailed study on secure hardware

Through implementing the projects, I have learned many details about secure hardware, and have tried to provide feedback to the smartcard community.

  – Development environment

I have added support for Cyberflex Access in the sc7816 library, i.e., loading and unloading applets, as well as DES and RSA key loading. This enabled smartcard software development on UNIX. Before this, we needed to use Windows. This is used at CITI on a daily basis, and was used in the smartcard seminar, EECS 598, in the Winter semester of 2000 at the University of Michigan.
– Performance studies

I have added support for performance evaluation in the sc7816 library. This was used in almost all performance evaluation sections in this dissertation. Using this tool, I have found interesting facts about smartcards, e.g., in Cyberflex Access, RSA is faster than DES for buffer larger than 32 byte (this is quite interesting, considering RSA is usually about 100 times slower than DES in software implementation), iButtons, which were believed faster than smartcards, are slower than Cyberflex Access in many operations. I have also found interesting things about secure coprocessors’ performance. IBM 4758 has a very fast bulk DES engine, but the engine is not so effective on small data.

– Problem findings

I have found many bugs and problems of smartcards. The lack of real DES functionality in many smartcards, which I reported in Section 3.1.6, is a representative example. Other examples include these findings in Cyberflex Access: DES initialization vector must be declared with 64 byte buffer, instead of 8 byte. Setting a DES key through setKey() method leaks memory.

6.3 Future Direction

Here are some future directions that should be explored.

• Deployment of the developed software

I believe secure hardware is very useful. Unfortunately, they have not yet been widely deployed. I believe this is partly due to the high cost of deploying
smartcard readers and smartcards. Some workstation vendors (e.g., SUN Microsystems) are planning to put smartcard readers into their workstations by default. This should make deployment easier.

I believe that once people start using secure hardware for some application, the next applications and the deployment of them will come quickly. This is why I have written many applications, hoping one of them become the killer application. I believe some of my work has demand for use, especially Kerberos / smartcard integration, SCFS, and SC-CFS. I will continue to deploy these applications.

- Porting software to Windows NT

I have written almost all software on either OpenBSD or Linux. I understand that Windows NT is a dominant operating system in the marketplace, and therefore it is essential to write software on NT if I want the software to be used by many people. I could not do much porting due to time limitation, but this should be done.

In particular, I was asked to port SCFS to NT several times. I implemented a preliminary version of SCFS on NT based on Framework for Implementing User-Mode File Systems in Windows NT, or FIFS [1], to prove this is possible, but could not finish it due to time limitation.

- Rewriting SCFS on NFS

SCFS is written based on Arla, which is a two-layer structured software. One part is in the kernel level, and the other in the user level. Because the major portion is in the user level, SCFS is much more portable than entirely kernel level filesystems. However, it is not as portable as being written entirely in the
user level, like CFS. CFS is extremely portable because it is written as a NFS daemon, where the NFS protocol Version 2 has not changed for years. SCFS will be much more portable if it is rewritten as a NFS daemon.

- **Smartcard access from BIOS**

  In the secure booting protocol, it is desirable to move the smartcard access library into BIOS. This would allow modification to hardware configuration without reprogramming the BIOS chip. I think this feature is essential for secure booting technique to be used.

- **Smartcard with I/O devices**

  As Schneier et al. [90] and Balfanz et al. [5] have pointed out, a smartcard based system suffers from the lack of trusted I/O path from a user to a smartcard. For example, when a user types a PIN, it goes through a host, on which a Trojan horse may be installed to steal it. On another example, when a user signs e-mail with his smartcard, he does not know what message the smartcard is really signing. The host may alter the message and tell the smartcard to sign it.

  These problems can be solved by an input device and an output device attached on a smartcard, respectively. This is a problem to be addressed, too.

- **Improvement of Secure Hardware RPC**

  I have developed Secure Hardware RPC in the Kerberos server / secure coprocessor integration project. This is a useful tool, but can be much more useful if it embraces a standard interface definition language, such as ONC RPC.

- **Migration from DES to DESX or Triple-DES**
All of my applications that involve symmetric key encryption use DES. This should be replaced by DESX or Triple-DES in the near future.
Appendix A
Published Papers

Most of the projects I have done were published as conference papers. Abstracts of the papers are attached here.
Practical Security Systems with Smartcards

Naomaru Itoi and Peter Honeyman
Center for Information Technology Integration
University of Michigan
Ann Arbor

itoi@eecs.umich.edu, honey@citi.umich.edu

Abstract: Secure hardware is a useful tool for enhancing computer system security. Traditionally, researchers have attempted to build secure operating systems by creating secure hardware and developing on top of it. Our approach is to integrate commodity secure hardware, i.e., smartcards, into existing operating systems.

This paper describes three projects aimed at practical secure operating systems based on smartcards: smartcard integration with Kerberos V5, a UNIX filesystem for smartcards, and Internet Protocol on smartcards. The first two are implemented and indicate satisfactory performance, while the last is under development.

Smartcard Integration to Kerberos V5

Naomaru Itoi and Peter Honeyman
Center for Information Technology Integration
University of Michigan
Ann Arbor

itoi@eecs.umich.edu, honey@citi.umich.edu

Abstract: We describe our design and implementation of smartcard integration with Kerberos V5. Authentication is among the most important applications for smartcards and is one of the critical requirements for computer security. By augmenting Kerberos V5 with tamper-resistant hardware, we enhance the security of Kerberos V5 and offer a potential “killer application” leading to wider adoption of smartcard technology.

SCFS: A UNIX Filesystem for Smartcards

Naomaru Itoi, Peter Honeyman, and Jim Rees
Center for Information Technology Integration
University of Michigan
Ann Arbor

itoi@eecs.umich.edu, honey@citi.umich.edu, rees@umich.edu

Abstract: Smartcard software developers suffer from the lack of a standard communication framework between a workstation and a smartcard. To address this problem, we extended the UNIX filesystem to provide access to smartcard storage, which enables us to use files in a smartcard as though normal UNIX files.

Secure Coprocessor Integration with Kerberos V5

Naomaru Itoi
Center for Information Technology Integration
University of Michigan
Ann Arbor
itoi@eecs.umich.edu

Abstract: The nightmare of Trusted Third Party (T3P) based protocol users is compromise of the T3P. Because the compromised T3P can read and modify any user information, the entire user group becomes vulnerable to secret revelation and user impersonation. Kerberos, the most widely used network authentication protocol, is no exception. When the Kerberos Key Distribution Center (KDC) is compromised, all the user keys are exposed, thus revealing all the encrypted data and allowing an adversary to impersonate any user. If an adversary has physical access to the KDC host, or can obtain administrator rights, KDC compromise is possible, and catastrophic. To solve this problem, and to demonstrate the capabilities of secure hardware, we have integrated the IBM 4758 secure coprocessor into Kerberos V5 KDC. As a result of the integration, our implemented KDC preserves security even if the KDC host has been compromised.


\footnote{This project has been carried out in the IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598, in the summer of 1999.}
Secure Internet Smartcards

Naomaru Itoi, Tomoko Fukuzawa, and Peter Honeyman
Center for Information Technology Integration
University of Michigan
Ann Arbor

itoi@eecs.umich.edu, tomokof@umich.edu, honey@citi.umich.edu

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 Java-Cards. 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.

Bibliography


160

[26] didnog@l0pht.com. any local user can gain administrator privileges and/or take full control over the system. L0pht Security Advisory, February 1999. http://www.l0pht.com/ advisories.html.


[45] Maurice Herlihy and Doug Tygar. How to make replicated data secure. In

[46] R. Hermann, D. Husemann, and P. Trommler. Opencard framework 1.1 pro-

In CARDIS’98, Louvain-la-Neuve, Belgium, Sept. 1998. Third Smart Card Re-
search and Advanced Application Conference.

Sidebotham, and M.J. West. Scale and performance in a distributed file system.

tocards.

[50] The International Organization for Standardization and The International Elec-
trotechnical Commission. ISO/IEC 7816-4 : Information technology - Identifi-
cation cards - Integrated circuit(s) cards with contacts, 9 1995.

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

Implementation of a provably secure, smartcard-based key distribution proto-
Research and Advanced Application Conference.


