SCFS: A UNIX Filesystem for Smartcards

Naomaru Itoi, Peter Honeyman, and Jim Rees
University of Michigan, Ann Arbor

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, but with the additional security properties inherent to smartcards.

1 Introduction

Today, it is easy to purchase smartcards in reasonable prices, *e.g.*, \$5 - \$20 for each. However, smartcard software development is hard: smartcard software developers have long suffered from the lack of a user friendly standard communication protocol between application software[1] and a smartcard. The ISO-7816 communication protocol [9] is so widely accepted that virtually all smartcards support it.[2] However, the 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 high-level interfaces such as UNIX files and I/O streams. This hampers our ability to build application software.
- Although all smartcards support ISO-7816, details of implementation of the protocol differs among vendors and types of smartcards. This requires software developers 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, *etc.* For example, the CLA byte of application class[3] is `0x00` in some smartcards (Giesecke & Devrient STARCOS Version 2.1), while it is `0xc0` in others (Schlumberger MultiFlex).

To address the deficiencies of ISO-7816, many new standards have been proposed. Examples are:

- General purpose standards: Open Card Framework (OCF) [2, 8] and PC/SC [3, 4].
- Special purpose standards: PKCS #11 [12] for cryptography, EMV [5] and SET for electronic commerce [13].
- On-chip software standards: JavaCard [15] and MULTOS [16].

Although these standards provide abstractions at a higher level than ISO-7816-4, it remains a

[1] "Application software" is a program running on a workstation that communicates with a smartcard. A program running on a smartcard is called "on-chip software".

[2] Almost all smartcards support ISO-7816-1, -2, and -3; many also support ISO-7816-4 [18]

[3] See Guthery and Jurgensen [6] or ISO-7816 [9] for a description of of "CLA" and "application class."

challenging task for developers to select 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.*, OCF limits the programming language to Java; PC/SC is used only with Windows – and create their own API dependencies, because software written for one standard does not run with another. We discuss these issues in Section 5.1.

Our solution to this problem is to embrace a classic, sophisticated API – the UNIX filesystem – instead of inventing a new one. The UNIX filesystem API suits a smartcard well because a smartcard is a passive device used for secure storage: a smartcard stores data (secrets), and responds to requests from a workstation to read or write the data. It does not initiate actions. This passivity is characteristic of storage devices such as hard disks. Cryptographic functions, such as get challenge, internal and external authenticate, verify key and PIN, are unique to smartcards. However, smartcards still act passively for these functions, and they are implemented with `ioctl()`.

In UNIX operating systems that support `vnodes` (equivalently, Virtual Filesystem, or VFS) [11] [14], it is possible to write a virtual filesystem that communicates with a special hardware device, *e.g.*, a smartcard, and mount it in the UNIX filesystem name space. The mounted hardware device then becomes identical to any UNIX filesystem hierarchy 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 `ls`, `cd`, `pwd`, and `cat`, and system calls such as `open`, `read`, and `write` on files in the smartcard.

We have implemented a smartcard filesystem (or SCFS) in the OpenBSD-2.4[4] kernel. With SCFS mounted, a user or an application can use files in a smartcard as she would normal UNIX files.

[4]OpenBSD is a free, 4.4BSD-based operating system. `http://www.openbsd.org`

The remainder of this paper is organized as follows. Section 2 describes our goals and the design of SCFS. Section 3 details implementation of SCFS. (Readers not interested in implementation details may want to skip Section 3.) Performance evaluation in Section 4 shows that the overhead of SCFS is small and does not substantially degrade the performance of smartcard software. We discuss SCFS with a comparison to other standards in Section 5. Future direction is described in Section 6 and concluding remarks are in Section 7.

2 Design

2.1 Design Goals

Our goal is to provide a user friendly interface to access a smartcard. We define design goals as follows, although not all can be achieved, for reasons outlined in Section 2.2:

- Files in a smartcard should be indistinguishable from other UNIX files.
- A smartcard can be accessed with any UNIX system call (*e.g.*, `creat`, `open`, `read`, and `write`).
- UNIX commands (*e.g.*, `ls`, `cd`, `pwd`, and `cat`) can be used to access files in a smartcard.
- The smartcard VFS must be able to access any smartcard that supports ISO-7816.
- The smartcard VFS should hide details about a smartcard to users.
- Security of a smartcard must be preserved.
- No smartcard files may be cached in the UNIX system because a smartcard is a more secure place to store data (see the end of Section 2.3).

2.2 Design Problems

A huge obstacle to achieving our goals is the absence of a standard way to request metadata information about files in a smartcard. Some information essential for the UNIX filesystem is simply not present in a smartcard, *e.g.*, 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.

We have two choices, with concomitant tradeoffs:

- Dictate an internal format on a smartcard to store information such as directory entries, length of a file, *etc.*, in a file in a smartcard. This provides full functionality of UNIX filesystems.
- Degrade functionality of SCFS. For example, no `ls`, no `cat`.

We compromise between the two choices. We 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. We 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 ISO-7816) in a smartcard has a file called `2e.69` ("`.i`") containing the requisite metadata.

2.3 Design

Inspired by Arla [19], SCFS is implemented as a kernel module, `xfs`, that handles VFS requests, and a user daemon, `scfsd`, that communicates with an ISO-7816 smartcard. Figure 1 shows the overview of the design.

When an application calls a VFS operation (*e.g.*, `read` or `write` to a smartcard file), the

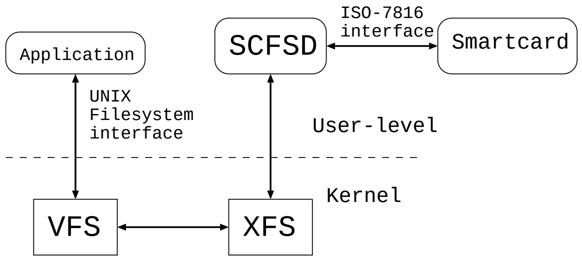

Figure 1: SCFS design

kernel module upcalls `scfsd` to request service. `Scfsd` creates ISO-7816 APDUs,[5] 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 [17] for handling the ISO-7816 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 knowledge of a smartcard before it is mounted, *e.g.*, existence of special APDUs, opcodes used for APDUs, ATRs[6] they return, *etc.* The information is stored in a SCFS configuration file, `/usr/scfs/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 3.6.

Unlike most UNIX filesystems, SCFS does not cache data read or written 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

[5] An *Application Protocol Data Unit*, or APDU, can be viewed as a framing protocol for messages passed from application software to a smartcard [9].

[6] Answer To Reset.

in a smartcard and is not protoected at all from an adversary with administrative privileges. In addition, files in a hard disk are usually backed up on tape, which may fall into the hands of an adversary.

3 Implementation

3.1 Overview

As described in Section 2.3, SCFS is separated into a kernel module (`xfs`) and a SCFS daemon (`scfsd`), detailed in Sections 3.2 and 3.3, respectively. Communication between `xfs` and `scfsd` is detailed in Section 3.4. Implementation of SCFS is based on Arla-0.6. Communication between `xfs` and `scfsd` is derived directly from Arla.

3.2 Kernel Module (xfs)

The kernel module (`xfs`) implements a virtual filesystem, the `ioctl` system call, and communication with `scfsd`.

The virtual filesystem consists of several functions called by the kernel when a file in SCFS is accessed. For example, the core part of the `read` system call is implemented by the `xfs_read()` vnode operation in the `xfs`.

We describe some important vfs operations, `xfs_mount()` and `xfs_root()`, and some important vnode operations, *i.e.*, `xfs_lookup()`, `xfs_read()`, `xfs_write()`, `xfs_getattr()`, and `xfs_readdir()`, in Section 3.5.

`Xfs` is typically loaded into the kernel at boot time. When `xfs` needs to communicate with a smartcard, it performs the communication by upcalling `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.

3.3 SCFS daemon (scfsd)

`Scfsd` performs operations requested by `xfs`. For requests that require smartcard access, `scfsd` translates the request to ISO-7816 APDUs. Figure 2 shows an example of message flow when an application requests to read 8 data bytes from a smartcard.

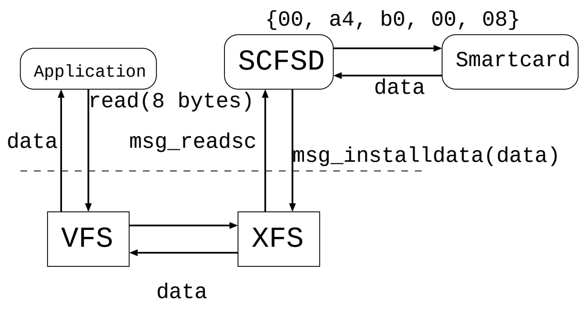

Figure 2: Reading 8 data bytes from a smartcard

3.4 Communication between xfs and scfsd

`Xfs` communicates with `scfsd` through RPC. When `xfs` needs access to a smartcard, 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:

```
struct xfs_message_readsc msg;

msg.header.opcode = XFS_MSG_READSC;
msg.buf = buf;
msg.size = 8;
msg.offset = 0;
fidcpy (msg.fid, xnode->handle);
xfs_message_rpc(fd, &msg.header,
                        sizeof(msg));
```

After invoking `xfs_message_rpc()`, `xfs` sleeps until it receives the result of the request. `Scfsd` eventually receives data from a smartcard and sends it back to the kernel module. Here is an example of sending a reply message:

```
struct xfs_message_installdata msg;

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

3.5 Important VFS/Vnode operations

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

VFS Operations:

- `Xfs_mount()` mounts SCFS on a specified directory. It first sends a reset signal to the smartcard. When it receives ATR from the smartcard, it scans the configuration file to find a smartcard description that matches the ATR, reads the configuration information, initializes `scfsd`, initializes `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 `fid`.[7] 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 components, 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, as follows. (1) It selects the target file. (2) 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. (3) With the file now selected, `xfs_read()` sends a read APDU (*e.g.*, `c0 b0 00 00 length`) to the smartcard. (4) If the read request fails because the file is protected by a PIN, `scfsd` prompts the user for a PIN. The prompt is directed to the controlling tty of the application that issued the system call. (5) Finally, `scfsd` passes the data read back to the user via a call to the `xfs` layer and kernel `uiomove()`.
- `Xfs_write()` behaves identically to `xfs_read()`, except for the direction of data.
- `Xfs_getattr()` 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()` is typically called by a `getdirentries()` 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 that we define our own metadata strategy, described in Section 2.2. `Xfs_readdir()` constructs full directory entries from the directory entries and from our metadata file and returns them to the application.

[7] 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`.

Some functionalities in a smartcard do not fit 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, we use the `ioctl()` operation.[8] `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 a key, internal authentication, external authentication, get response, and get challenge APDUs.

3.6 Configuration File

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

```
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
}
```

[8] We use `ioctl()` to avoid adding a new system call; this decision will be revisited someday.

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 Performance Evaluation

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

4.1 Method

We measured total elapsed time and smartcard access time for various vnode operations. The difference reflects filesystem overhead. Figure 3 shows this relation.

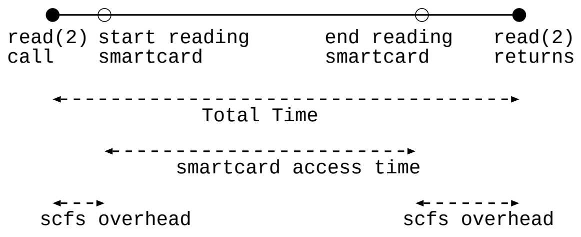

Figure 3: Performance Evaluation

Serial communication with smartcards uses 12 bits per byte (one start, eight data, one parity, two stop bits). Our test harness communicates with MultiFlex at 38.488 Kbps, or 312 μsec. per byte, and with CyberFlex Access at 55.928 Kbps, or 215 μsec. per byte.

4.2 Result

To measure read and write performance, we used six different operand sizes, ranging from 1 byte to 254 bytes. Figure 4 shows the result. For both cards, elapsed time as a function of operand size is very close to linear.

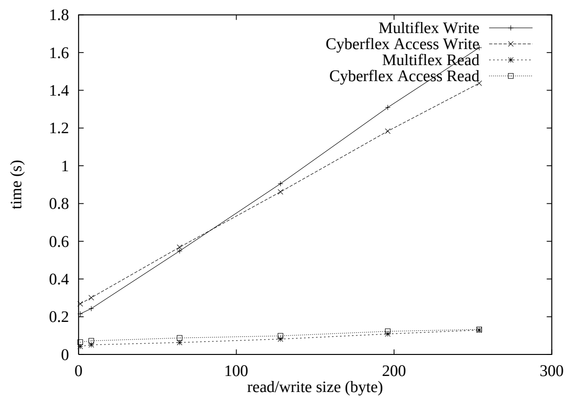

Figure 4: read/write time

Table 1 shows the card-related per-byte cost for these functions. Most of the overhead, evident by the non-zero y-intercept, is due to (unknown) card processing time; operating system overhead is under 1 ms in each case. Some of the per-byte card processing time is evident in the difference between the theoretical minimum read cost and the measured cost. Write times are substantially longer than the theoretical minimum, reflecting the time required to write to EEPROM.

Card	Op	Per-byte
MF	read	0.345
CFA	read	0.257
MF	write	5.62
CFA	write	4.62

Table 1: Read and write performance. All times in ms.

4.3 Breakdown

Table 2 shows the cost of some other operations for the CyberFlex Access card.

Op	Card	OS	Bytes
open	0	.566	0
lseek	0	0.339	0
create	466	2.00	78
remove	626	37.4	64
vrfykey	258	1.67	10

Table 2: Read and write performance. Times in ms. Bytes represent the amount of data transfered in the operation.

The directory structure cache, created at mount time by reading the "`.i`" file, is evident in the open and lseek operations, which do not communicate with the card. Verify key, not a vnode operation, is implemented with pioctl.

5 Discussion

5.1 Related Work

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

OCF

OpenCard Framework is middleware that supports a smartcard with Java [2, 8] by providing high-level APIs, vendor transparency, card type transparency, and extensibility. These objectives are similar to ours. 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" [7]. 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);
      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();
      } catch (Exception e) {
        e.printStackTrace(System.err);
      }
    }

    System.exit(0);
  }
}
```

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 can also be viewed as a disadvantage. Java and its object oriented model modularize and simplify complex software, but a smartcard is a simple, passive device. For many smartcard applications, Java might be viewed as overkill.

In SCFS, we use a smartcard in a simple way. For example, we can print out a file (as in the OCF example) by typing:

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

OCF cannot be used with languages such as C and C++, the languages in which most operating systems and security protocols are written. Consequently, OCF offers little to enhance directly the security of many operating systems and security protocols, such as UNIX, Kerberos, SSH, and PGP.

PC/SC

PC/SC is a general purpose architecture for integrating a smartcard into PCs [3]. Its objectives are similar to OCF and SCFS. According to part 6 of the specification [4], the PC/SC API is similar to the UNIX filesystem, featuring Open(), Close(), Read(), Write(), Seek(), *etc.* Therefore, usability of PC/SC and SCFS are similar.

Unlike OCF, PC/SC supports multiple languages and development environments, such as C, VC++, VB++, and Java. However, it is used only with Windows operating systems. While SCFS currently supports only OpenBSD, it is possible to port it to other UNIX systems, and (perhaps) even to Windows NT.[9]

[9]If we purchase the Installable Filesystem package.

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. We may provide libraries for specific purposes to wrap around SCFS to give higher abstractions.

5.2 Advantage of SCFS

Transparent API with the UNIX Filesystem

SCFS differs from the other approaches such as OCF and PC/SC because it is implemented as an operating system extension. Consequently, to an application, smartcard files look identical to files stored on other media. With SCFS, an application can use a smartcard without modification (Figure 5).

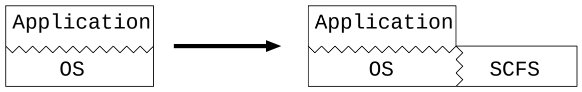

Figure 5: Application is not modified to use SCFS.

With SCFS, many UNIX applications can take advantage of smartcard security without modification. For example, here is how we made SSH work with a private key stored in a smartcard: we 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
```

Although not tested yet, 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 6).

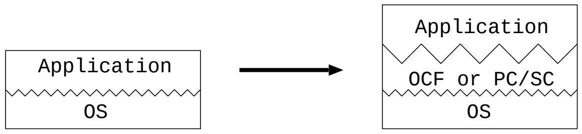

Figure 6: Application must be modified to use OCF or PC/SC

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.

5.3 SCFS as 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, *etc.* Depending on the type of applications, different kinds of abstraction may be required. Therefore, there are many standards that do not interoperate [1]. 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 allows users to access a smartcard with sophisticated UNIX commands, such as `cd`, `ls`, `pwd`, `cat`, *etc.* SCFS is especially helpful in maintenance, testing, and debugging; Figure 7 depicts our model of SCFS as a development tool.

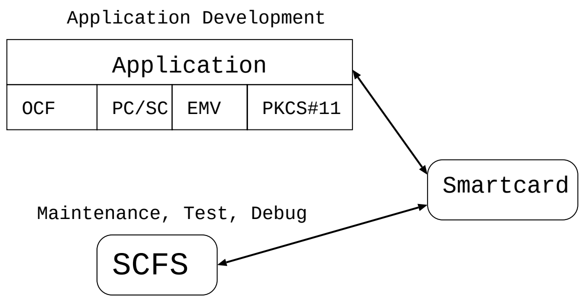

Figure 7: SCFS as a low-level development tool.

6 Future Directions

Some ideas derived from other smartcard standards suggest enhancements to SCFS. In PC/SC, a smartcard specific driver is loaded as a DLL (Dynamic Loadable Library). In SCFS, smartcard specific code is directly written in the user-level daemon, `scfsd`. PC/SC's approach is more extensible than ours because it does not require recompilation to add a driver for a new smartcard. We are considering extending SCFS to have the same advantage with dynamically loadable libraries.

We intend to port SCFS to different operating systems and to support more smartcard types. (Currently it supports only Schlumberger MultiFlex, CyberFlex, and PayFlex).

Security of SCFS should be explored. SCFS is currently vulnerable to Trojan Horse attacks, *i.e.*, if an adversary has administrative privileges, she can install a rogue version of SCFS that steals a user's PIN or modifies contents of the smartcard. We are investigating integrity checking and authentication of SCFS code by a smartcard.

We plan to use SCFS in several applications. One of them, storing Kerberos tickets, is particularly interesting, as it dovetails with our related Kerberos V5 smartcard extensions [10]. In that application, the smartcard performs decryption on Kerberos tickets. Storing the result in a protected SCFS file indicates the synergy of our approach.

7 Conclusion

We 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 to develop smartcard software. Performance evaluation shows the overhead caused by SCFS is negligible.

8 Acknowledgment

We thank the Arla developers: Assar Westerlund, Love Hornquist-Astrand, Magnus Holmberg and many more people in the arla-drinkers mailing list for patiently answering our questions.

This work was supported by Schlumberger's Program in Smartcard Technology at CITI.

References

[1] Duncan W. Brown. Application development: A new focus for smart

card suppliers and implementaters. In *CardTech/SecureTech'98*, volume 1, pages 352–353, Washington, DC, April 1998.

[2] OpenCard Consortium. General information web document, Oct. 1998. http://www.opencard.org / docs / gim/ ocfgim.html.

[3] Microsoft Corporation. Smart cards, white paper, April 1998. http://www.microsoft.com/ smartcard/ smartcards/ scardwp.asp.

[4] PC/SC Workgroup (Microsoft Corp. etc.). Interoperability specification for ICCs and personal computer systems, part 1-8, December 1997. http://www.smartcardsys.com.

[5] Europay, MasterCard, and Visa. EMV'96: Integrated circuit card application specification for payment systems, June 1996. http://www.mastercard.com/ emv/emvspecs02.html.

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

[7] R. Hermann, D. Husemann, and P. Trommler. OpenCard framework 1.1 programmer's guide, Oct 1998. http://www.opencard.org / docs / pguide / PGuide.html.

[8] Reto Hermann, Dick Huseman, and Peter Trommler. The OpenCard framework. In *CARDIS'98*, Louvain-la-Neuve, Belgium, Sept. 1998. Third Smart Card Research and Advanced Application Conference.

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

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

[11] S. R. Kleiman. Vnodes: An architecture for multiple file system types in sun unix. In *Proceedings of USENIX Summer Technical Conference.* USENIX, 1986.

[12] RSA Laboratories. PKCS #11: Cryptographic token interface standard. version 2.01, December 1997. http://www.rsa.com / rsalabs / pubs / PKCS/.

[13] SET Secure Electronic Transaction LLC. SET standard technical specifications, 1999. http://www.setco.org/.

[14] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley Publishing Company, 1996.

[15] SUN Microsystems. Java card technology. http://java.sun.com:80 / products / javacard/ index.html.

[16] MULTOS. http://www.multos.com/.

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

[18] James F. Russell. Compatibility and conflicts: PC/SC, OCF, Java card, MULTOS ... In *CardTech/SecureTech'98*, volume 1, pages 97–101, Washington, DC, April 1998.

[19] Assar Westerlund and Johan Danielsson. Arla - a free AFS client. In *Proceedings of USENIX 1998 Annual Technical Conference*, pages pp. 149 – 152, New Orleans, Louisiana, USA, June 1998. USENIX. http://www.stacken.kth.se / projekt / arla.