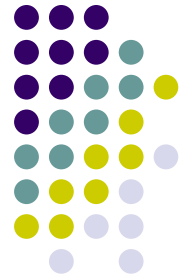


Stackable FileSystems

Seminarbeitrag von Frank Tobian
für das Seminar Dateisysteme





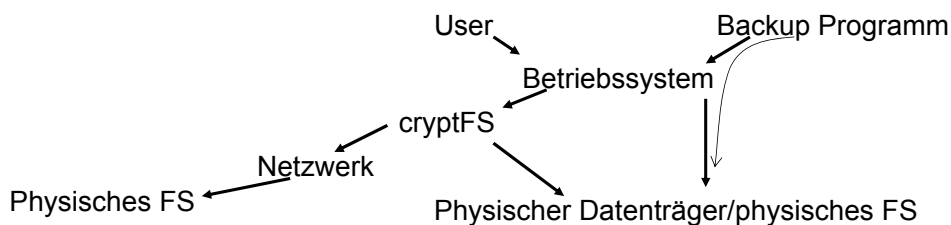
Überblick

- **Was und wozu sind Stackable FS**
 - Was ist ein Stackable FS?
 - Wozu dienen sie?
- WrapFS – ein Template
- FiST – A FileSystem Translator language
- Beispiele

Stackable FS - Was



- stapelbare Dateisystemsichten
- symmetrische und unabhängige Schnittstellen
- fan-out – mehr als ein Mount unter dem DS z.B. UnionFS
- fan-in – Zugriff auf tiefere Schichten.



Keine Stapelspeicherverwaltung – wie man vielleicht zuerst annimmt
Aufeinander stapelbare Layer die unabhängig voneinander arbeiten.

Input = Output (symmetrisch) – Option fürs durchschleusen von „unbekannten“
Befehlen

Stackable FS können mehrere FanIns und FanOuts haben

Fan-in ist für den User nützlich, um zB Backups zu machen, er kann so direkt auf die verschlüsselten Daten auf Platte zugreifen, und so auch direkt die verschlüsselten Daten sichern, ohne sie vorher entschlüsseln zu müssen.

Ansonsten müssen Read Befehle nicht durch eine Verschlüsselungsschicht und Write Befehle nicht durch die Entschlüsselungsschicht.

Fan-out kann mehrere Dateisysteme darunter verwalten, sei es ein Netzwerkbackup, UnionFS oder ein selbst geschriebenes Raid, das z.B. die Daten auf Veränderungen beim lesen überprüft.

Stackable FS – Wozu?



- (Klassische) Dateisysteme werden in Kernelcode geschrieben
- Einfach erweiterbar
- Einfaches Debuggen
- Einfacher und schneller zu programmieren
- Userspace zu ineffizient (overhead > 10%)
- Teilweise portabel (WrapFS und FiST)

4

Kernelcode, sehr kompliziert - Verstehen von C Code, Kernel Code, Kernel Funktionen, alles sehr komplex

Userspace(code) einfacher und portabler, aber mindestens 10% Performance Verlust (viele Quellen sprechen von 20 bis 30%)

Für neue Funktionen kopiert man im „klassischen“ Sinn ein anderes System, dessen Quellcode man kennt, versucht ihn zu verstehen und fügt die neue Funktion hinzu. Über 90% der Zeit geht im Normalfall für reines Verstehen drauf. Dies ist sehr ineffizient und schreckt ab.

Bei Stackable muss man den Code des zugrunde liegenden FS nicht mal kennen. Könnte auch closed source sein, so lange es die Schnittstellenstandards einhält

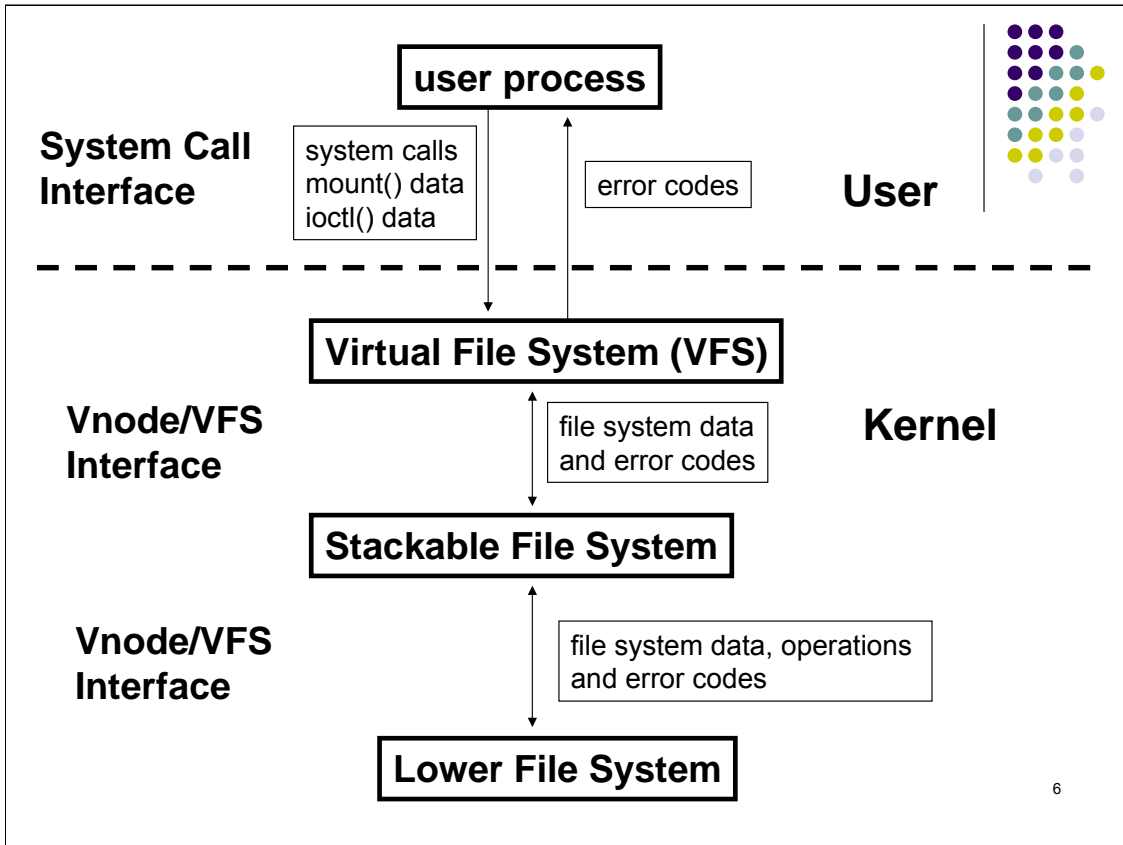
Code der mit WrapFS und FiST geschrieben wurden, kann Linux Solaris und BSD Code erzeugen, vielleicht auch bald WinNT

Die Weiterentwicklung von klassischen Dateisystemen erfordert extrem viel Zeit und ist meistens nur durch Unterstützung großen Firmen möglich, das testen dauert Jahre und wenn es mal läuft wird es ungern geändert, auch ein Grund für Stillstand in der Weiterentwicklung

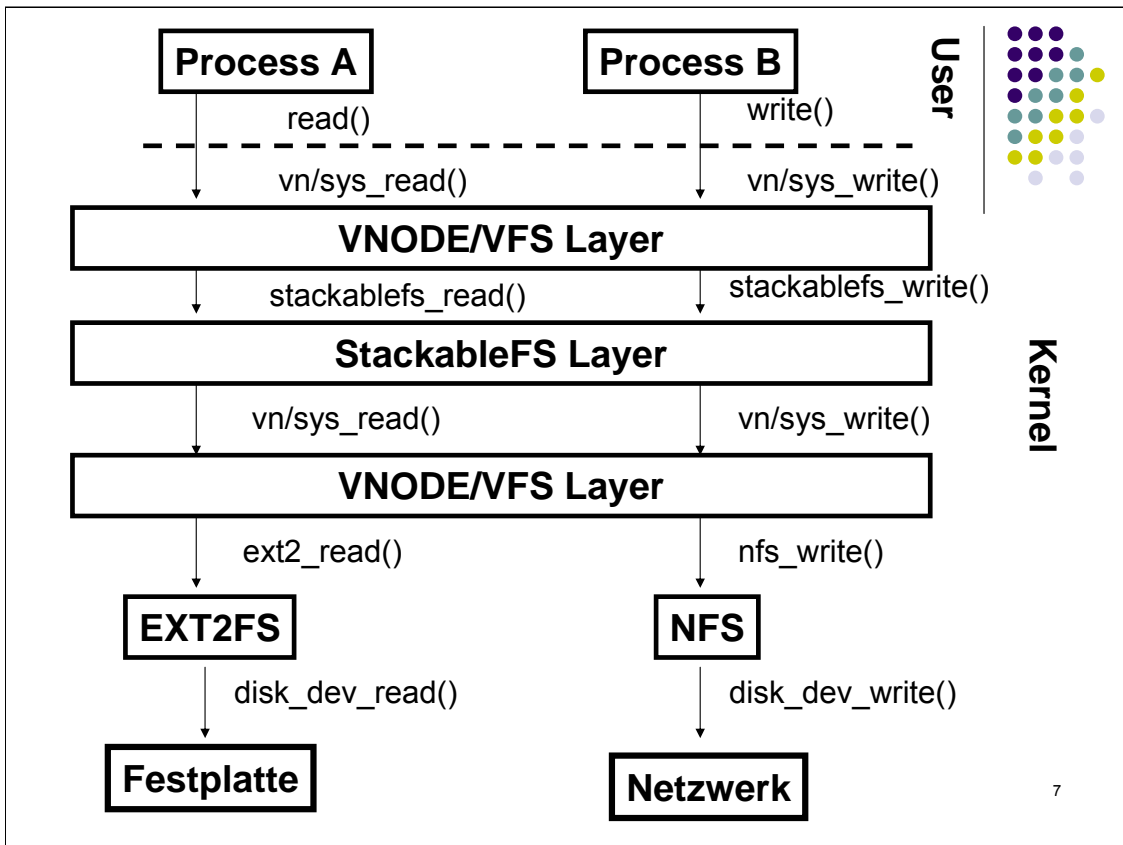
Klassische Dateisysteme



Medientyp	Klassische FS	Code Größe (C Code Zeilen)
Harddisk	UFS,FFS,EXT	5000 - 20000
Network	NFS	6000 - 30000
CD-ROM	HSFS, ISO9660	3000 - 6000
Floppy	PCFS,MS-DOS	5000 - 6000



Passt nicht zum 2.6er Linuxkernel, vielleicht 2.0, sonst BSD



Passt nicht zum 2.6er Linuxkernel

BSD besitzt Vnodes und macht `vn_read`

Linux nutzt VFS und macht `sys_read`



Überblick

- Was und wozu sind Stackable FS
- **WrapFS – ein Template**
 - WrapFS Überblick
 - Aufrufchnittstellen
 - mounten
- FiST – A FileSystem Translator language
- Beispiele



WrapFS

- Einfache Vorlage, die Schnittstellen zum Kernel anbietet
- Erzeugt Kernel Code (Modul) dadurch gute Performance
- Nutzt bereits existierende Dateisysteme
- Kann Metadaten und Daten manipulieren oder durchschleusen

WrapFS – Aufrufchnittstellen



- `encode_data` – Daten schreiben
- `decode_data` – Daten lesen
- `encode_filename` – Dateiname schreiben
- `decode_filename` – Dateiname lesen
- Keine spezielle Attributsmanipulation

10

`Encode_data` wird aufgerufen bei jedem Schreibprozess, durch die API kann man hier die Daten manipulieren

`Decode_data` wird aufgerufen bei jedem Leseprozess, zum Beispiel um verschlüsselte Daten zu entschlüsseln, muss nicht bei jedem FS selbst geschrieben werden

Analog `encode` und `decode filename`, bei Dateinamen muss man noch auf illegale Zeichen prüfen und die manchmal ein weiteres man kodieren, sodass z.B. kein / im Dateinamen auftaucht.

Besonderes: Doppertes Caching, einmal kodiert und einmal unkodiert

Dateiattribute werden nur über `ioctl` manipuliert. Limitierte Möglichkeiten wegen `VNode` Interface

Hier eine API zu erzwingen würde die Performance wahrscheinlich zu stark negativ beeinflussen

2 Dinge in `WrapFS` greifen auf die, und diese werden oft aufgerufen, z.B. (lookup)

WrapFS – mount



- Normales mounten: 2 Pfade
Ein Mountpunkt (/mnt) und ein Verzeichnis auf das WrapFS aufsitzt (/usr)
mount -t wrapfs /mnt /usr
Zugriff auf /usr/ucb ist ohne WrapFS
Zugriff auf /mnt/ucb geht durch WrapFS
- Overlaymount (Zugriff nur mit WrapFS)
mount -t wrapfs -0 /usr

11

Wenn unser WrapFS nun verschlüsseln würde, so könnten Programme (wie Backup) direkt auf /usr/ucb die verschlüsselten Daten zugreifen, Zugriffe über /mnt/ucb werden durch WrapFS wieder entschlüsselt, bevor sie ans Programm weitergegeben werden.

Bei Overlay mount kann man nur mit WrapFS bzw. durch die Stapel auf die Daten (Festplatte) zugreifen



Überblick

- Was und wozu sind Stackable FS
- WrapFS – ein Template
- **FiST – A FileSystem Translator language**
 - Überblick
 - Aufbau der Grammatik
 - fistgen
- Beispiele



FiST

- FileSystem Translator Language
- eigene „Sprachmodule“
- Templates:
 - BaseFS{0,1,2,3}
 - BaseFS3 ohne Zusatzcode = WrapFS

FiST Grammatikaufbau



```
%{  
1 C Declarations  
%}  
2 FiST Declarations  
%%  
3 FiST Rules  
%%  
4 Additional C Code
```

14

Erst der C Kopf eingeschlossen in %{ }%

C headers, define macros or typedefs, list forward function prototypes, etc.
Gelten für den gesamten Rest des Codes

FiST Declarations

read-only or not, whether or not to include debugging code, if fan-in is allowed or not, and what level (if any) of fan-out is used.

special data structures, mount-time data that can be passed with the mount system call,

FiST can also define new error codes: an encryption file system can return a new error code indicating

FiST Rules greift direkt auf die Vnode Befehle zu um sie zu modifizieren bzw. eine eigene Schicht (Layer, stackable) dazwischen zu schalten, siehe WrapFS und VFS Vortrag

fistgen



FiST Input File



BaseFS Templates



fistgen



Stackable File System Sources

15

Fistgen generiert aus der Fist Input Datei und einem BaseFS Template den Stackable FS Code

Aus diesem Code lassen sich dann loadable Kernelmodule erstellen

Wenn ein Programmierer filter:data im FiST input file deklariert verlangt fistgen in der C Code Sektion zwei Funktionen, encode_data und decode_data.



Überblick

- Was und wozu sind Stackable FS
- WrapFS – ein Template
- FiST – A FileSystem Translator language
- **Beispiele**
 - SnoopFS
 - UsenetFS
 - CryptFS
 - UnionFS

SnoopFS



- Speichert unberechtigte Zugriffe auf Dateien
- Funktionsweise:
 - Fehler bei lookup abfangen
 - Testen ob `uid == root` oder `uid == owner`
 - Falls nicht:
 - Protokolliere unberechtigten Zugriff
 - Sonst ist der Zugriffsfehler für den Eigentümer nicht von Bedeutung



SnoopFS Code

- FiST Code:

```
%op: lookup: postcall {  
  if ((fistLastErr() == EPERM ||  
      fistLastErr() == ENOENT) &&  
      $0.owner != %uid && %uid != 0)  
    fistPrintf("snoopfs detected access  
by uid %d, \  
pid %d, to file %s\n", %uid, %pid,  
$name); }
```

UsenetFS



- Usenet hat viele tausende Dateien in einem Ordner
dies ist auf normalen DS langsam
- UsenetFS teilt automatisch jedes manuell erstellte Verzeichnis in 1000 (beliebig viele) Unterverzeichnisse
- Sortieren nach Vorletzter bis 4. letzter Stelle

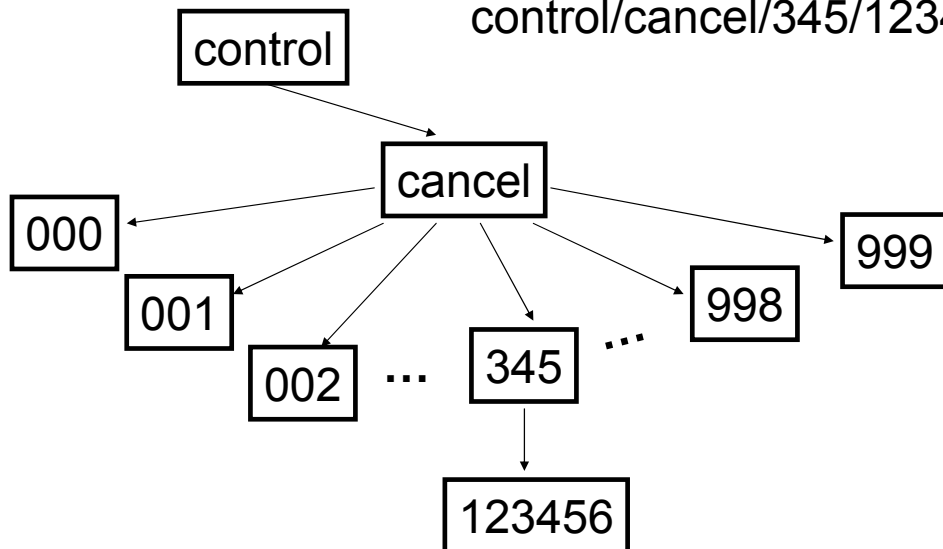
We therefore distribute articles across 1000 directories named 000 through 999. Since article numbers are sequential, we maximize the distribution by computing the final directory into which the article will go based on three lesser significant digits, skipping the least significant one. For example, the article named control/cancel/123456 is placed into the directory control/cancel/345/.

UsenetFS



- control/cancel/123456 →

control/cancel/345/123456



CryptFS

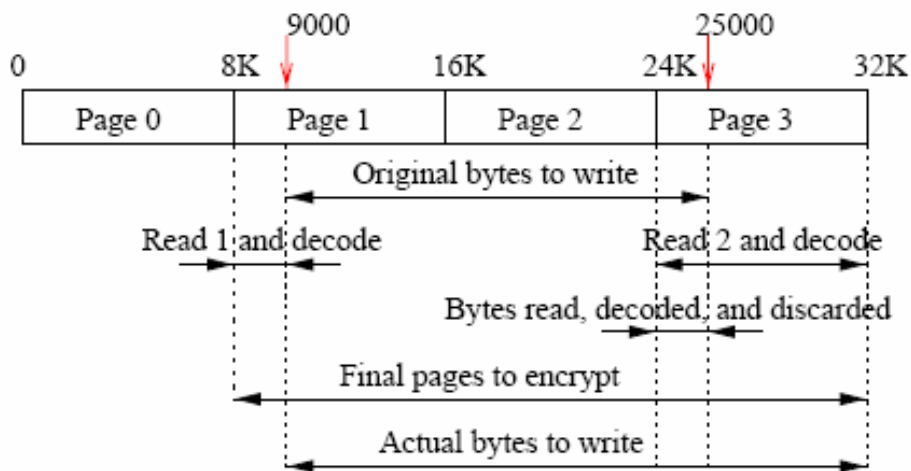


Figure 2: Writing Bytes in Cryptfs

21

Funktionsweise von ecryptfs und cryptfs sind sehr ähnlich, da letzte Woche ecryptfs vorgestellt wurde, heute zu cryptfs nur Tabellen ;)
Solaris loopback file system (lofs).

Zur Erinnerung: CryptFS arbeitet auf Pages, sodass, wenn eine Datei nicht an den Anfang einer Page steht, diese erst entschlüsselt werden muss. Cipher Block Chaining (CBC) wird genutzt

Teil des CryptFS FiST Codes:

```
%{
#include <blowfish.h>
%}
filter:data;
filter:name;
ioctl:fromuser SETKEY {char ukey[16];};
per_vfs {char key[16];};
%%
%op:ioctl:SETKEY {
char temp_buf[16];
if (fistGetIoctlData(SETKEY, ukey, temp_buf)<0)
fistSetErr(EFAULT);
else
BF_set_key(&$vfs.key, 16, temp_buf);
}
%%
unsigned char global_iv[8] = {
0xfe, 0xdc, 0xba, 0x98, 0x76, 0x54, 0x32, 0x10 };
int cryptfs_encode_data(const page_t *in, page_t *out)
{
int n = 0; /*blowfish variables */
unsigned char iv[8];
fistMemCpy(iv, global_iv, 8);
BF_cfb64_encrypt(in, out, %pagesize, &($vfs.key),
iv, &n, BF_ENCRYPT);
return %pagesize;
}
...

```

CryptFS



File System	Writes		Reads		1024× readdir
	1024× 8KB	8× 1MB	1024× 8KB	8× 1MB	
ext2fs	3.33	3.06	0.17	0.34	1.49
lofs	3.40	3.35	0.30	0.34	1.51
wrapfs	3.48	3.58	0.18	0.34	1.57
cryptfs	9.27	8.33	0.26	0.34	3.18
nfs	26.85	17.67	0.47	3.17	16.27
cfs	101.90	50.84	0.89	8.77	118.35
tcfs	110.86	84.64	6.45	7.94	34.83

Table 1: Linux x86 Times for Repeated Calls (Sec)

Wie man hier und anhand der folgenden Tabellen sieht, sind stackable FS nicht langsamer als non stackable ;)



File System	Writes		Reads		1024× readdir
	1024× 8KB	8× 1MB	1024× 8KB	8× 1MB	
ufs	4.88	3.98	0.48	0.38	0.52
cryptfs	63.95	11.10	10.72	7.23	7.14
nfs	54.17	18.82	1.69	0.38	0.28
cfs	140.78	140.98	27.68	24.57	18.02

Table 2: Solaris x86 Times for Repeated Calls (Sec)

File System	Writes		Reads		1024× readdir
	1024× 8KB	8× 1MB	1024× 8KB	8× 1MB	
ufs	12.55	6.04	1.00	1.01	0.15
cryptfs	56.59	22.55	1.04	1.05	0.29
nfs	55.69	21.63	1.31	1.09	0.33
cfs	99.34	31.80	2.09	4.80	0.87

Table 3: FreeBSD x86 Times for Repeated Calls (Sec)

UnionFS



- read-only oder read-write
- Reihenfolge mit eindeutigem „Rang“
- fanout n; im FiST Deklarationsteil
- Bei read-write muss oberster Mount beschreibbar sein
- copy on write

24

Bei read-write muss oberste Ebene beschreibbar sein, damit man darauf schreiben kann, wenn man eine Datei von readonly editieren möchte

Fanout n: Anzahl der zugrunde liegenden Partitionen

Copy on write kopiert die Daten von read only medien auf die nächst höhere read write schicht, damit sie dort geschrieben werden können.

Beispielcode:

```
fanout 2;
%%
%op:lookup:postcall {
if (fistLastErr() == ENOENT)
fistSetErr(fistLookup($dir:2, $name));
}
%op:readdir:postcall {
fistSetErr(fistReaddir($dir:2, NODUPS));
}
%delops:all:postcall {
fistSetErr(fistOp($2));
}
%writeops:all:call {
fistSetErr(fistOp($1));
}
```

UnionFS



- delete
 - whiteout
 - all
 - first
- snapshoting
- sandboxing

25

delete=whiteout (default) findet die erste Instanz der Datei und löscht diese. Setzt ein Whiteout, dass die Datei als gelöscht markiert

delete=all findet jede Instanz der Datei und löscht sie alle

delete=first findet die erste Instanz der Datei und löscht diese, sollte die Datei auf einer tieferen Ebene existieren, so wird diese Instanz der Datei nun sichtbar (die nächst höhere Ebene, sollte es noch mehr Instanzen geben)

Snapshots: Schalte alle Ebenen auf read only und mounte als oberste Ebene eine Beschreibbare hinzu, alle Änderungen geschehen nur auf der obersten Ebene

Sandbox: wenn ein potentiell gefährlicher Prozess entdeckt wird und man macht ein Snapshot, muss der Administrator ALLE Änderungen prüfen, wenn man aber die alten Prozesse einfach normal weiterarbeiten lässt und nur für den potentiell gefährlichen mit der Snapshot Technik von Veränderungen des aktuellen Systems ausschließt, muss ein Administrator nur diese Änderungen nachprüfen



Quellen

- Prof. **Erez Zadok**
mit der Homepage <http://filesystems.org/>
Fast alles was man mit Google über Stackable Filesystems findet ist von Erez Zadok oder es bezieht sich auf seine Veröffentlichungen
- Robert Jankovics [Paper](#) (Student TU Wien)

Wer sich einlesen will <http://filesystems.org/docs/zadok-phd-thesis/thesis.pdf> ist ein Allumfassendes Paper